Prof. Edgar Serna M. (Ed.)

Métodos Formales, Ingeniería de Requisitos y Pruebas del Software

Editorial Instituto Antioqueño de Investigación

2021 Medellín - Antioquia

Prof. Edgar Serna M. (Ed.)

Métodos Formales, Ingeniería de Requisitos y Pruebas del Software

ISBN: 978-958-53278-0-1

Medellín, Antioquia

Catalogación en la publicación – Biblioteca Nacional de Colombia

Serna M., Edgar
Métodos formales, ingeniería de requisitos y pruebas del software [recurso electrónico] / Edgar Serna M. ed. -- 1a. ed. -- Medellín: Instituto Antioqueño de Investigación, 2021.
Archivo en formato digital (pdf). -- (Ingeniería y ciencia)

Incluye referencias bibliográficas al final de cada capítulo.

ISBN 978-958-53278-0-1

1. Ingeniería de software 2. Ingeniería de requerimientos 3. Programas para computador - Pruebas I. Título II. Serie

CDD: 005.30287 ed. 23 CO-BoBN– a1068583

Investigación Científica
ISBN: 978-958-53278-0-1
DOI:
Hecho el Depósito Legal Digital

Métodos Formales, Ingeniería de Requisitos y Pruebas del Software
Serie: Ingeniería y Ciencia
Editorial Instituto Antioqueño de Investigación

Edición 1: febrero 2021
Publicación electrónica gratuita

Maquetación: Instituto Antioqueño de Investigación IAI
Diseño: IAI, Medellín, Antioquia.
Editorial Instituto Antioqueño de Investigación es Marca Registrada del *Instituto Antioqueño de Investigación*. El resto de marcas mencionadas en el texto pertenecen a sus respectivos propietarios.

Diseño, edición y publicación
Editorial Instituto Antioqueño de Investigación
http://fundacioniai.org/index.php/editorial.html

Instituto Antioqueño de Investigación IAI
http://fundacioniai.org
contacto@fundacioniai.org

CONTENIDO

PRESENTACIÓN

En el ciclo de vida del desarrollo del software destacan dos fases como las más importantes y con mayor impacto en la Ingeniería del Software: la Ingeniería de Requisitos y las Pruebas del Software. Si bien en los procesos con los que actualmente se desarrolla software las separan como fases independientes, la realidad es que están ligadas íntimamente. Los equipos de desarrollo consideran que las Pruebas solamente se pueden diseñar e implementar cuando el producto está terminado, pero si la meta es entregarles a los clientes y usuarios software fiable, seguro y de calidad, tienen que verla como una fase paralela a todo el ciclo de vida, y diseñarla e implementarla en este sentido.

La Ingeniería de Requisitos amerita mención aparte, porque si esta fase no se estructura y aplica de forma organizada, las demás fases del ciclo de vida tendrán problemas. En esta fase se descubre, elicita y documenta las necesidades de los clientes y usuarios, y su producto, la especificación de requisitos, constituye la base para que el equipo de desarrollo diseñe y lleve a cabo las demás fases. Por eso no es de extrañar que la comunidad de las Ciencias Computacionales la considere como lo más importe de la Ingeniería de Requisitos.

Cuando el equipo de trabajo aplica adecuadamente las técnicas de elicitación, de documentación y de especificación de requisitos, entrega un documento creíble y de calidad para continuar con las demás fases. Ese documento es básico, porque contiene la descripción de las necesidades que el sistema debe satisfacer, tanto las funcionales como las no-funcionales, sobre las que se diseña la arquitectura del software y la estructura de las bases de datos, y entrega información importante para determinar cuestiones como el lenguaje de programación y el diseño del plan de pruebas.

La cuestión es que las técnicas de elicitación se aplican y documentan en lenguaje natural, con los inconvenientes de ambigüedad y de interpretación que esto genera. Documentar los requisitos de esta forma genera inconvenientes y procesos de reingeniería para el equipo de desarrollo, porque debe traducir esos requisitos al lenguaje matemático con el que trabajan los computadores, pero no tienen la claridad suficiente de lo que necesitan los clientes y usuarios. Por eso es que se recomienda la utilización de los Métodos Formales, los cuales sirven para unificar, desde la elicitación, el lenguaje con el que dialogan las partes interesadas y con el que se documentan los requisitos.

La tecnología de los Métodos Formales surgió como una especie de salvavidas para ayudar a resolver la llamada crisis del software. El asunto es que para nadie es un secreto que somos una sociedad software-dependiente y que este producto impregna cada vez más las actividades humanas. Pero dada la creciente complejidad de los problemas que se pueden resolver con software y la baja capacidad profesional de los programadores, su calidad y fiabilidad no responden a esa responsabilidad. Una posible solución es matematizar la Ingeniería de Requisitos, porque las matemáticas es un lenguaje universal, no tiene ambigüedades y hace parte de los procesos formativos en todas las disciplinas.

El objetivo de este libro es presentar una serie de aportes, productos de investigación, en cada una de las áreas descritas: los Métodos Formales, la Ingeniería de Requisitos y las Pruebas del Software. La idea es reunirlos en un solo texto para que los equipos de desarrollo los puedan consultar con facilidad, a la vez que tener a la mano las teorías y modelos que pueden aplicar con la intención general de mejorar la calidad y fiabilidad del software.

El libro es sí es una recopilación de los trabajos del profesor Edgar Serna y su equipo, quienes han dedicado gran parte de su investigación a aportar al logro de la meta de mejorar los productos software, y que, si bien han sido publicados en revistas o presentados en Congresos, reunirlos en un texto completo es una apuesta para que los ingenieros de software tengan herramientas más cercanas. Se trata de una iniciativa en la que se reúne la producción de estos investigadores a partir de un programa de investigación estructurado y ejecutado por más de dos décadas.

Esperamos que el libro sirva como fuente de consulta y que se comparta en la academia, la industria y los investigadores, porque la intención es que todos nos unamos en un trabajo conjunto para mejorar la calidad y fiabilidad de los productos software. El nuevo orden mundial, al que entraremos pronto, estará marcado por una dependencia generalizada del software. Por eso es apremiante comenzar a mejorar la forma en que se estructura, desarrolla y libera, porque la sociedad estará al compás de cómo responde este desarrollo tecnológico, y podría convertirse en una línea gris de la cual dependerá nuestra supervivencia como especie.

PRIMERA PARTE

MÉTODOS FORMALES

CAPÍTULO I
Los Métodos Formales en contexto[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

La investigación alrededor de los métodos formales ha progresado en calidad y volumen en los últimos años; las herramientas y las ideas relacionadas se utilizan cada vez más en la industria y en la ciencia en una amplia variedad de formas; muchos de los problemas desafiantes en la construcción de sistemas software seguros, en la programación de procesadores multi-núcleo y en los sistemas ciber-físicos requieren el apoyo formal para su modelado y análisis, y en los procesos de investigación y aplicación de las Ciencias Computacionales y en otras disciplinas existe un número creciente de aplicaciones. Pero a pesar del creciente interés y de su necesidad e importancia, todavía no constituyen una parte integral de los procesos formativos convencionales en pregrado y posgrado. En este capítulo se resume algunos de los desafíos que enfrentan los métodos formales para lograr mayor utilización y aplicación en los procesos formativos y en la industria, y se describe las oportunidades para su aplicación tradicional y no-tradicional.

[1] Presentado en la XIII Conferencia Iberoamericana en Sistemas, Cibernética e Informática CISCI. Orlando, USA. 2014.

INTRODUCCIÓN

Para explicar por qué funcionan las cosas o por qué fallan se utiliza argumentos formales, conformados por una estructura precisa de axiomas y reglas de inferencia, y para verificar su correctitud estos argumentos formales se comprueban mecánicamente mediante la codificación de los pasos de deducción. En muchos casos los argumentos también se pueden generar mecánicamente, y también se utilizan técnicas automatizadas de razonamiento para generar ejemplos de algunas limitaciones matemáticas.

Recientemente, ha habido un incremento significativo en la investigación de los métodos formales, lo que ha hecho posible un sinnúmero de aplicaciones de amplio alcance para esta tecnología, que se pueden aplicar para apoyar un riguroso y coherente plan de estudios en matemáticas y en Ciencias Computacionales. Además, los métodos formales para verificar hardware, algoritmos y protocolos, y el modelado y el diseño de sistemas complejos, se pueden incorporar en los cursos de otras sub-disciplinas de las Ciencias Computacionales. En la industria se pueden aplicar en una amplia variedad de formas, ya sea para elicitar requisitos y reglas de negocios, para generar casos de prueba y escenarios, para explorar diseños, para verificar protocolos, o para producir prototipos seguros. En otras disciplinas científicas y de ingeniería se pueden utilizar para apoyar el modelado y el análisis a nivel de sistema.

En las próximas décadas los métodos formales encontrarán una audiencia mucho más amplia, cubriendo una vasta gama de disciplinas. Este panorama debe servir para que se incorporen en los planes y procesos formativos, en lo que actualmente tienen poca o ninguna referencia. El objetivo es contar a corto plazo con profesionales capacitados para que, mediante representaciones formales, interpreten, modelen y presenten soluciones eficientes y eficaces a los problemas. Es un hecho que el desarrollo de la tecnología presentará retos cada vez más complejos, y los métodos formales son la instancia a la que se puede recurrir para presentarles soluciones más fiables [1].

1. MARCO REFERENCIAL

La creciente complejidad de la información es un desafío para la investigación en el siglo XXI, por ejemplo, un automóvil moderno tiene entre 70 y 80 procesadores operando una amplia gama de funciones sofisticadas, por otro lado, el componente software cuesta cerca de la mitad del valor del desarrollo de sistemas, y su fiabilidad impacta las economías de organizaciones y países. Además, se acrecienta la crisis de la ingeniería, cuando el interés por tomar estudios en estas áreas, o en las científicas, como las Ciencias Computacionales, decrece cada año [2]. Por ejemplo, la Oficina de Estadísticas Laborales de los Estados Unidos estima un crecimiento constante en la oferta laboral en Ciencias Computacionales, pero las matrículas en pregrado en esta misma área decrecen a medida que avanza el siglo. Es evidente que hay bastante por hacer en el asunto de estas ciencias como disciplina dinámica, intelectualmente rica y con crecientes vínculos con otras disciplinas científicas y de ingeniería.

Las Ciencias Computacionales y las matemáticas se basan en el estudio, la construcción, la comprensión y la aplicación de la lógica y la abstracción puras, y aunque los computadores existen en la realidad física, la computación se encarga de la representación y la manipulación de información, un componente no-físico. Además, mientras que las matemáticas tratan con las leyes de las entidades abstractas, como los números, los conjuntos, las superficies, los volúmenes, las relaciones, el álgebra y los homomorfismos, la computación se focaliza en la manipulación de la información, representada en abstracciones como los bits, los vectores-bit y las estructuras de

datos. Es decir, las matemáticas se utilizan para representar y analizar la realidad física, y la computación para modelar y simular esa realidad. Mientras que las primeras trabajan con operaciones y descripciones concisas, el hardware y el software son construcciones más complejas, y se espera que se desempeñen con precisión y fiabilidad, incluso al trabajar con datos poco fiables [3].

Un científico computacional debe desarrollar ampliamente su capacidad lógico-interpretativa y abstractiva, de tal forma que pueda comprender los problemas y modelar soluciones con el objetivo de formalizar requisitos; construir representaciones eficaces y expresivas de datos; diseñar algoritmos correctos y eficientes que funcionen con todas las entradas posibles; y moldear esos diseños en un código confiable [4]. La lógica y la abstracción son esenciales para formalizar el entorno en el que el producto software funcionará, para especificar los requisitos, para diseñar e implementar la arquitectura del sistema y los componentes, y para depurarlo y mantenerlo en el tiempo. Estos científicos están constantemente solucionando problemas, generando representaciones de datos, algoritmos y protocolos, diseñando nueva programación y lenguajes descriptivos, y resolviendo conflictos de sincronización. En los últimos sesenta años, desde las Ciencias Computacionales se ha desarrollado un cuerpo de conocimientos lógico-abstracto para resolver estos problemas.

Los sistemas computacionales tratan la complejidad en todos los niveles, desde las plataformas hardware, los protocolos y las representaciones de datos, hasta los sistemas ciber-físicos y las sofisticadas amenazas a la seguridad. Los métodos formales representan un enfoque para la gestión de esta complejidad, mediante el uso de representaciones simbólicas de abstracciones extraídas de las matemáticas y la computación a través de fórmulas lógicas, que se perciben como obstáculos a resolver, pero que se pueden manipular de varias maneras para construir pruebas. Por otro lado, la lógica se puede utilizar para demostrar que un conjunto de restricciones implica a otro. Por esto, la computación trata con fenómenos dinámicos, autómatas y máquinas de estado, y la lógica temporal representa formalismos expresivos para capturar el comportamiento y las propiedades computacionales.

Los métodos formales se desarrollan desde hace más de cincuenta años, y han introducido principios y paradigmas como la programación funcional, la programación lógica, los lenguajes de especificación, el álgebra de procesos, la interpretación abstracta, la verificación en tiempo de ejecución, y una serie de otras innovaciones conceptuales influyentes. Casi una cuarta parte de los premios Turing son reconocimientos a trabajos con un significativo componente de métodos formales. Pero, a pesar de todo, esta área del conocimiento todavía parece estar en una encrucijada: por un lado, en lo positivo, esta tecnología está bien desarrollada y se sustenta en un amplio número de aplicaciones y de usuarios y desarrolladores industriales importantes; sin embargo, en el lado negativo, ha dejado de ser un componente principal para los procesos formativos en Ciencias Computacionales e ingeniería.

Son pocos los profesores e investigadores trabajando en el área, la oferta de cursos a nivel de pregrado y postgrado es escasa, difícilmente se desarrolla proyectos importantes, y solo un pequeño número de graduados e investigadores los acoge como fuente de trabajo. Los planes de estudios parecen no desarrollar adecuadamente la capacidad lógico-interpretativa y abstractiva de los estudiantes, que por el contrario parece haber disminuido en los últimos años, y cada vez tienen más aversión por las matemáticas [5].

Aun así, el tema de los métodos formales es emocionante y cada vez más relevante. Con el dramático incremento en el poder computacional de las últimas décadas es el momento de llevar

la computación a la esfera semántica de los modelos y de la inferencia. Este cambio posibilitará una serie de nuevas aplicaciones en los procesos formativos e industriales. Como ejemplo se pueden citar los modelos de hardware digital y analógico, las teorías matemáticas, los sistemas físicos estáticos y dinámicos, las redes biológicas, los sistemas software complejos, e incluso a la psicología, la filosofía y las interacciones sociales.

La formación, en todos los niveles, se puede beneficiar del modelado y del análisis formal, y del uso más profundo de metáforas computacionales [6]. Actualmente, la formación en informática se desarrolla en términos de tareas de procesamiento de información de bajo nivel, mientras que la percepción real se obtiene mediante el uso de modelos computacionales y de Ciencias Computacionales de alto nivel. Es el caso de los autómatas finitos, que pueden servir como una metáfora útil para una serie de tareas, como el funcionamiento de una máquina expendedora, los analizadores de lenguajes, o los videojuegos. Los juegos interactivos constituyen otra metáfora computacional que se utiliza frecuentemente en los procesos formativos, y los sistemas dinámicos se pueden utilizar para modelar flujos continuos, como los vehículos en movimiento, los proyectiles, los niveles de fluidos, los reguladores de velocidad, los circuitos analógicos y los péndulos; los sistemas híbridos combinan cambios de estado discretos y continuos, y se pueden utilizar para termostatos modelo, sistemas de control de navegación y sistemas biológicos y económicos. Casualmente, estos modelos computacionales, metáforas y tecnologías pueden enriquecer los procesos formativos en todos los niveles, porque apoyan la formación, el desarrollo de habilidades, la colaboración y la comunicación.

La aceptación industrial de los métodos formales también se puede ampliar. Actualmente, hay industrias productoras y consumidoras de herramientas formales, por ejemplo, Intel y AMD las utilizan para controlar la equivalencia de circuitos, generar secuencias de prueba, controlar modelos de controladores de estado finito, y demostrar teoremas acerca de algoritmos aritméticos de punto flotante; Microsoft tiene una serie de proyectos que utilizan software de comprobación de modelos y razonamiento automatizado, para analizar el código secuencial para terminación, correctitud asercional, y en la eliminación de vías y callejones sin salida en sus procesos; Rockwell-Collins utiliza el razonamiento automatizado en la verificación de software; empresas de diseño electrónico automatizado, como Synopsys, Cadence y Mentor Graphics, venden herramientas para el control de modelos y equivalencias, y el verificador de diseño Simulink de Matlab puede generar casos de prueba y verificar propiedades; y no se puede dejar de mencionar la utilización que NASA hace de los formalismos en todas las simulaciones para los vuelos espaciales.

Sin embargo, estos esfuerzos todavía son relativamente modestos en comparación con la posibilidad de uso sistemático de herramientas formales en todos los procesos [7]. Es cierto que la tecnología formal todavía puede no estar lista para un uso masificado, pero en este momento se tiene un marco para pensar en aplicaciones más ambiciosas y no-tradicionales, y es cuando las universidades y sus procesos formativos deben cobrar relevancia [8].

Por todo lo anterior, se necesita innovaciones audaces que aprovechen la potencia de las herramientas y técnicas formales modernas, para transformar radicalmente los procesos formativos, tanto en Ciencias Computacionales como en otras disciplinas ingenieriles que aplican metáforas computacionales, y se pueden aprovechar en el modelado asistido por computador y en el diseño de sistemas complejos. Si se cultiva adecuadamente este momento en la historia, se podrán iniciar trabajos con el objetivo de mejorar la calidad y la fiabilidad de los productos software.

2. CONTEXTO DE LOS MÉTODOS FORMALES

La idea de colocar a las matemáticas y a la ciencia sobre una base axiomática se remonta a Pitágoras, Platón, Aristóteles y Euclides. El uso de un lenguaje formal, soportado en razonamiento mecanizado, fue defendido por primera vez por Raymon Lully en el siglo XIII, y más tarde por Gottfried Leibniz en el siglo XVII. Vannevar Bush sugirió, refiriéndose al uso de la tecnología computacional para gestionar grandes volúmenes de información, que la lógica podría llegar a ser muy difícil, y que sin duda sería muy conveniente generar más seguridad en su uso, y predijo que algún día sería posible hacer *clic* sobre las opciones operativas de una máquina, con la misma seguridad con la que se opera una caja registradora [9].

En el siglo XXI, con el incremento de la potencia computacional y el desarrollo de potentes herramientas software para el razonamiento formal, la ciencia está mucho más cerca de alcanzar los objetivos que se trazaron estos pensadores visionarios. La necesidad de la formalización es especialmente aguda en protocolos que implican concurrencia y criptografía, y sin el rigor que otorga sería extremadamente difícil lograr con certeza que estos protocolos alcancen sus propiedades sin un punto muerto, un bloqueo activo, o ciertas condiciones de prueba. La semántica formal explica la interpretación matemática de un trozo de texto del programa o de una propiedad, y sus postulados son técnicas de razonamiento acerca de los programas y sus propiedades. Entonces, aplicar exitosamente los métodos formales se trata de un pre-requisito y su carácter composicional es un desafío clave en la semántica formal, porque es conveniente poder desarrollar las propiedades de los programas en términos de sub-programas constituyentes [10]. Los recientes desarrollos en semántica posibilitan la formalización del comportamiento de los programas con almacenamiento dinámico, Programación Orientada por Objetos, y programas concurrentes [11].

Por otro lado, la automatización juega un papel significativo en la aplicación exitosa de los métodos formales. En la Verificación del comportamiento de software y hardware de estado finito, y con una cantidad limitada de esfuerzo, se puede utilizar la Verificación formal de modelos para anotar los estados con sus propiedades de comportamiento. Para una determinada propiedad, esto se puede hacer en tiempo lineal en el tamaño del espacio del estado del sistema. Por supuesto, este espacio puede ser amplio, porque incluso un sistema con n bits tiene 2^n estados posibles. El problema de la explosión de estados en la Verificación formal se ha abordado mediante el uso de representaciones simbólicas, abstracciones y reducciones de orden parcial, y también se ha extendido a sistemas de estado infinito, como los sistemas paramétricos, los sistemas de tiempo real y los sistemas híbridos lineales. Mediante la Verificación limitada de modelos la satisfacibilidad booleana también se utiliza para representar y razonar acerca de los sistemas de estado finito.

En las últimas décadas se ha presentado importantes mejoras algorítmicas para estos procesos de satisfacibilidad, que también han sido utilizados para las restricciones que proceden de una mezcla de teorías, como las de la aritmética, las matrices, los vectores bits, y los tipos de datos recursivos. Estos procedimientos se pueden utilizar para comprobar que una fórmula compleja no se puede satisfacer, o para generar contra-ejemplos, casos de prueba o calendarios, cuando la fórmula, en efecto, se puede satisfacer.

3. DESAFÍOS DE LOS MÉTODOS FORMALES

Hoy en día no debería existir mucha necesidad de debatir el hecho de que los métodos formales se deben utilizar más ampliamente en la industria del software. Muchos de los problemas de

complejidad que se presentan en la Ingeniería del Software, como los procesadores de múltiples núcleos, las máquinas virtuales múltiples, los requisitos de seguridad y los sistemas distribuidos globalmente, no se manejan adecuadamente mediante los métodos existentes, porque nos son suficientemente eficientes y eficaces en las pruebas. Además, la formación en Ciencias Computacionales e ingeniería se podría beneficiar con la introducción de herramientas y técnicas formales, sin embargo, existe muchos retos que se deben abordar antes de poder tener un caso convincente de sus beneficios:

1. Los lenguajes de programación para desarrollar software a gran escala tienen una semántica formal compleja, y parte de ella es innecesaria e inútil. Las herramientas formales tendrán que hacerle frente a la complejidad de otras características, como el flujo de información, los interruptores, los hilos y la concurrencia.

2. La aplicación exitosa de los métodos formales requiere habilidad e ingenio humano, incluso con una amplia automatización. Los usuarios potenciales podrían no estar dispuestos a invertir tiempo y dinero en una tecnología, si antes no tienen una idea clara de sus beneficios.

3. El proceso de la aplicación de los métodos formales en un proyecto de desarrollo de software puede ser oneroso, y todavía no es claro hasta qué punto su aplicación pueda reducir los costos de las pruebas.

4. Las herramientas de los métodos formales no son lo suficientemente amigables para un uso más amplio, en particular, todavía no tienen respuestas útiles cuando falla un esfuerzo de Verificación. Además, el mantenimiento de un desarrollo formal es una tarea de grandes proporciones, comparable a la del mantenimiento del software.

5. Existe diversas herramientas formales poderosas, y los problemas difíciles requieren frecuentemente una combinación de ellas. La interoperabilidad es un reto para estas herramientas, porque persisten diferencias semánticas sutiles que se debe solucionar.

Algunos de estos desafíos se están abordando al interior de la comunidad de investigadores en el área, y gran parte de la tecnología se puede cruzar con una funcionalidad transparente para el usuario [12]. El costo general de la aplicación de los métodos formales bajará en la medida que las herramientas se vuelvan más poderosas y más integradas. La ley de Moore predice una fuerte aceleración en el poder de las herramientas formales, y a medida que esta tecnología se estabilice, el desarrollo de interfaces de usuario formales se convertirá en una prioridad.

4. OPORTUNIDADES DE LOS MÉTODOS FORMALES

4.1 En los procesos formativos

Los métodos formales se deben integrar plenamente en los procesos formativos relacionados con las Ciencias Computacionales y algunas ingenierías. Cursos como complejidad computacional, análisis algorítmico, bases de datos, sistemas operativos, arquitectura de software, lenguajes de programación, Inteligencia Artificial y Análisis Numérico se pueden ilustrar con mayor eficacia mediante el uso de modelos, el análisis de métodos y las herramientas formales. Pero actualmente se imparten de forma tradicional y, en muchos casos, los currículos en Ciencias Computacionales se alejan cada vez más de las matemáticas, lo que puede influenciar en el hecho de que los egresados no estén significativamente mejor calificados para trabajar en computación, que aquellos con títulos en otras disciplinas. Existen algunos esfuerzos preliminares en esta

dirección, como el uso de la herramienta Alloy, que está dando resultados para formar en Ingeniería del Software formal, la herramienta Coq proof se está utilizando en algunas universidades para formar en meta-teoría de lenguajes de programación, la propuesta Wing [13] puede servir como modelo para incorporar métodos formales y herramientas de Verificación en cursos de Ciencias Computacionales, y el proyecto TeachScheme! experimenta con el demostrador de teoremas ACL2, para formar en desarrollo de software en pregrado [11].

Otras disciplinas también se pueden beneficiar de la utilización de los métodos formales. Este es el caso de los estudiantes de ingeniería eléctrica, que gracias al éxito de la Verificación de hardware están cada vez más expuestos al uso del razonamiento booleano y de la lógica temporal, y de muchos cursos de ingeniería que utilizan Matlab para el modelado y la simulación. Estos modelos se pueden formalizar, y podrían ofrecer ventajas pedagógicas, a partir de la integración del razonamiento formal con la simulación. Mathematica es una herramienta popular para el álgebra computacional, y un medio para desarrollar cursos [7, 12], que al igual que otros sistemas de álgebra computacional se puede mejorar con soporte para el desarrollo de pruebas, como se ha hecho con REDLOG [10]. La combinación del modelado, la construcción de pruebas y las herramientas de álgebra computacional pueden sustituir a los libros de texto por *live-books*, que combinen información con software para apoyar la experimentación, la visualización y las pruebas. Además, las herramientas de abstracción que se aplica en los cursos pueden hacer parte de la bibliografía de los estudiantes.

Una cuestión clave para incrementar el trabajo y la investigación en métodos formales es lograr el desarrollo de la capacidad lógico-interpretativa y abstractiva de los estudiantes. Sin esta capacidad puede que no abandonen la *fobia* por las matemáticas, o por las carreras con alto contenido de éstas. Además, es necesario que las instituciones estructuren e implementen procesos formativos en los que los profesores vinculen la teoría con la práctica, a través de proyectos más cercanos a la realidad. De esta forma, los estudiantes se desempeñarán mejor en la resolución de problemas, una de las funciones primordiales de los profesionales para el siglo XXI [4, 6, 14].

4.2 En la ciencia

El modelado formal y las técnicas de análisis se están utilizando en algunas ciencias para lograr mayor comprensión y entendimiento de los sistemas. Por ejemplo, las redes reguladoras de genes se pueden modelar como redes booleanas, sistemas híbridos, o sistemas estocásticos, y sus propiedades se pueden analizar utilizando la deducción y la Verificación de modelos y, para obtener una visión a nivel de sistemas de los diferentes procesos naturales y artificiales, se cuenta con la propuesta de Systems X. Los modelos a nivel de sistemas capturan abstracciones reales del programa destino utilizando máquinas de estado y limitaciones. Las técnicas formales se pueden utilizar para deducir las propiedades de estos modelos a nivel de sistema, y en el análisis aplicar teorías de prueba y de diagnóstico de problemas, lo mismo que para hacer predicciones.

4.3 En la industria

En Microsoft existe cerca de un centenar de investigadores que trabaja en métodos formales, y otras compañías tecnológicas, como Intel, AMD, Rockwell Collins y MathWorks, están desarrollando y utilizando activamente las herramientas formales. Los proveedores de diseño electrónico para automatización, como Cadence, Mentor Graphics y Synopsys, tienen importantes productos de Verificación formal en sus suites de herramientas. En Microsoft existe una decena de grandes proyectos para desarrollar herramientas formales de Ingeniería del Software, para

especificación, anotación, generación de casos de prueba, verificación de modelos y comprobación de seguridad, y muchos se centran alrededor de un potente motor para deducir y explorar modelos simbólicos. En los contextos con fiabilidad estricta y con requisitos de seguridad existe claramente una necesidad de herramientas, sin embargo, las más poderosas se pueden insertar en entornos de desarrollo integrado y en compiladores.

Un desafío particular para los métodos formales es en el desarrollo de sistemas ultra-grandes, que integran componentes software y hardware de diversas fuentes. Con la creciente aceptación de las capacidades de las herramientas formales, y la necesidad de requisitos más estrictos en fiabilidad y garantía, se puede esperar un incremento importante en la implementación y utilización de los métodos formales en la industria, a la vez que los laboratorios de investigación industrial podrán aportar importantes resultados y prototipos.

5. CONCLUSIONES

Las últimas décadas se pueden considerar como la época dorada de la investigación en métodos formales. El trabajo de los investigadores generó avances significativos en lógica, semántica de programas, lenguajes de especificación y de programación, metodologías de diseño, técnicas de análisis estático, razonamiento automatizado, herramientas de Verificación de modelos, y asistentes de prueba interactivos.

En las próximas décadas el enfoque de los métodos formales se centrará en ampliar su alcance y audiencia. Para lograrlo, deben ser lo suficientemente potentes para que se puedan utilizar sin ningún tipo de esfuerzo intelectual manifiesto, y buscar que la complementariedad entre la intuición y el juicio humano, y el cálculo y la inferencia por computador, se puedan explotar como una herramienta clave en la gestión de la complejidad de la información.

Además, se espera que los métodos formales no solo enriquezcan las Ciencias Computacionales, sino a muchas otras disciplinas e ingenierías que se pueden beneficiar de la formalización y la inferencia. El objetivo es que muy pronto se pueda hacer realidad el sueño de Leibniz de que: *en lugar de recurrir a argumentos técnicos, podamos ser capaces de sentarnos frente a un computador y decir: vamos a calcular*.

REFERENCIAS

[1] Serna E. (2010). Formal Methods and Software Engineering. Rev. Virtual Universidad Católica del Norte 30, 158-184.

[2] Serna E. y Serna A. (2013). Is it in crisis engineering in the world? A literature review. Rev. Fac. Ing. Univ. Antioquia 66, 197-206.

[3] Tiwari A. et al. (2003). Invisible formal methods for embedded control systems. Proceedings of the IEEE 91(1), 29-39.

[4] Serna E. (2013). Logic in Computer Science. Revista Educación en Ingeniería 8(15), 62-68.

[5] Tucker A. et al. (2001). Our curriculum has become Math-Phobic! En Thirty-second Technical Symposium on Computer Sciense Education. New York, USA.

[6] Serna E. (2015). La Capacidad Lógico-Interpretativa y Abstractiva. Fondo Editorial ITM.

[7] Scott S. (1991). Exploration with Mathematica. En R. Rashid (Ed.), CMU Computer Science: A 25th Anniversary Commemorative (pp. 505-519). Addison-Wesley.

[8] Boyer R. y Moore J. (1991). MJRTY – A fast majority vote algorithm. En R. Boyer (Ed.), Automated Reasoning: Essays in Honor of Woody Bledsoe (pp. 105-117). Kluwer.

[9] Bush V. (1945). As We May Think. Recuperado: http://www.theatlantic.com/ magazine/archive/ 1945/07/as-we-may-think/ 303881/.

[10] Dolzmann A. y Sturm T. (1997). REDLOG: Computer algebra meets computer logic. ACM Special Interest Group on Symbolic and Algebraic Manipulation 31(2), 2-9.
[11] Eastlund C. et al. (2007). Acl2 for freshmen: First experiences. En Seventh International Workshop on the ACL2 Theorem Prover and its Applications. Austin, USA.
[12] Kaltofen E. (1997). Teaching computational abstract algebra. Jour. Symb. Comput 23(5-6), 503-515.
[13] Wing J. (2000). Weaving formal methods into the undergraduate computer science curriculum. En 8th International Conference, AMAST 2000. Iowa, USA.
[14] Serna E. (2011). Abstraction as a critical component in Computer Science training. Rev. Avances en Sistemas y Computación 8(3), 79-83.

CAPÍTULO II
Perspectiva y aplicación de los Métodos Formales[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

En este capítulo se describe algunos aspectos relevantes de los métodos formales, se hace una descripción de su perspectiva histórica, de las definiciones ampliamente aceptadas, de sus aplicaciones y beneficios, y del marco del futuro de la investigación en el área. Es el resultado de una reflexión acerca de este componente de las Ciencias Computacionales y de la necesidad de tener una comunidad más amplia y sólida de investigadores, dedicada a fomentar y aplicar los métodos formales en la industria, pero también para que la academia los vea como un tema de formación en pregrado y posgrado.

[1] Presentado en la XIII Conferencia Iberoamericana en Sistemas, Cibernética e Informática CISCI. Julio 15-18. Orlando, USA. 2014.

INTRODUCCIÓN

Desde mediados del siglo XX los investigadores empezaron a trabajar y promocionar los métodos formales como la mejor alternativa disponible para desarrollar sistemas digitales seguros y confiables. Para muchos de ellos la eficacia, o más exactamente la necesidad, de los métodos formales está más que demostrada. Meyer opina que es claro, para las mentes más brillantes de las Ciencias Computacionales, que se necesita un enfoque matemático para desarrollo de software, si se quiere que progrese mucho más en calidad y fiabilidad [1]. Pero a pesar de esta audaz afirmación, la actitud de algunos ingenieros en ejercicio ha sido muy diferente, con una inclinación más hacia el rechazo que a su aceptación.

Aunque la situación ha cambiado un poco en el siglo XXI, especialmente dentro de la comunidad del diseño de hardware [2], la aceptación y el uso regular de los métodos formales todavía es mucho menor de lo que pensaron los pioneros. Los investigadores y los profesionales en esta área han tratado de comprender y explicar las causas de estas circunstancias [3-6], y algunos sugieren que se debe a la falta de herramientas adecuadas, falta de formación en matemáticas, incompatibilidad con las técnicas actuales, altos costos, falta de interés desde los procesos formativos, falta de decisión de los defensores, y poca o ninguna capacidad lógico-interpretativa y abstractiva de los ingenieros de software.

A pesar de llegar a conclusiones diferentes estos aportes tienden generalmente a abordar el tema de forma similar [2, 7]. Cada uno ha tratado de determinar por qué los ingenieros y científicos computacionales no utilizan habitualmente las actuales técnicas y herramientas formales. La suposición común parece ser que, aunque se ha demostrado suficientemente que la idea es buena, el problema radica en la aceptación de los detalles, no en la idea. Que los defensores de los métodos formales compartan esta suposición no es sorprendente, sin embargo, la reticencia de muchos ingenieros y científicos para utilizar o apoyar el desarrollo de cualquier método o herramienta formal sugiere otra posibilidad, y tal vez el problema de aceptación no está en los detalles, sino en la forma como la idea ha sido comunicada a los profesionales en sus procesos formativos.

1. ALREDEDOR DE LOS MÉTODOS FORMALES

El concepto *Métodos Formales* se refiere a técnicas y herramientas *matemáticamente rigurosas* para especificar, diseñar, Validar y Verificar sistemas software y hardware [2, 7]. *Matemáticamente rigurosas* significa que la especificación utilizada en los métodos formales está conformada por enunciados bien formados en una lógica matemática, y la Verificación formal por deducciones rigurosas en la misma lógica; es decir, cada paso sigue una regla de inferencia y, por lo tanto, se puede comprobar mediante un proceso mecánico. El informe FM89 [5] describe que estos métodos suman el rigor matemático al desarrollo, el análisis y la operación de sistemas computacionales y a sus aplicaciones, y que no son más que la concentración de las matemáticas aplicadas, en este caso la lógica formal, al diseño y al análisis de sistemas software intensivos.

En general existe una tendencia en la comunidad de los métodos formales para definirlos en términos de una axiomatización, al estilo Hilbert. Para Bloomfield [9] son métodos basados en un sistema matemático para especificar y producir software, que comprenden:

1. Una colección de notaciones matemáticas que direccionan las fases de especificación, diseño y desarrollo de software.

2. Un sistema de inferencia lógica bien fundada en el que se puede formular pruebas de Verificación formal y de otras propiedades.

3. Un marco metodológico en el que se puede desarrollar software a partir de una especificación, pero verificable formalmente.

La razón primordial para desarrollar técnicas formales es poder construir un marco dentro del cual se pueda predecir, científicamente, el comportamiento de los productos software. Mientras que las técnicas que subyacen en los casos estudiados hasta ahora son peldaños en pro de lograr una disciplina científica-ingenieril, los procesos para desarrollar sistemas controlados por computador están todavía en evolución. Por otro lado, es necesario aclarar que los términos *métodos formales* y *Verificación de programas* no son sinónimos, porque la Verificación es un componente de los métodos formales, mediante el cual se verifica que el software es compatible con la especificación. Este proceso lo describen Gries [10], Hoare [11] y Dijkstra [12], y se soporta en sistemas como el Gypsy Verification Environment [13].

Los métodos son *formales* en el sentido de que son lo suficientemente precisos como para ser implementados en un computador. Con esta tecnología es posible desarrollar especificaciones y modelos que describan parte o todo el comportamiento de un sistema, en varios niveles de abstracción, que se puede utilizar como entrada para un demostrador de teoremas automatizado. Para la Ingeniería de Requisitos la entrada puede ser una colección de especificaciones parciales y la salida un reporte sobre inconsistencias; para el diseño la entrada puede ser una especificación y una etapa de diseño, y la salida puede ser *1*, cuando la etapa de diseño es consistente con la especificación, o *0*, en otro caso; mientras que para la Verificación la entrada puede ser una especificación y una propiedad deseada del comportamiento del sistema, y la salida puede ser *1*, cuando la propiedad es una consecuencia de la especificación, o *0*, en otro caso.

El valor de los métodos formales radica en que proporcionan un medio para examinar simbólicamente todos los estados de un diseño digital, hardware o software, y para establecer una propiedad segura de correctitud, que sea verdadera para todas las posibles entradas [2]. Sin embargo, en la práctica y debido a la complejidad de los sistemas reales, rara vez se hace, excepto en los componentes de los sistemas de seguridad crítica. Existen diversos enfoques utilizados para trabajar con el tamaño y la complejidad de los estados asociados a los sistemas reales:

- Aplicar métodos formales para elicitar y especificar los requisitos y para los diseños de alto nivel, donde la mayoría de los detalles quedan ocultos.
- Aplicar métodos formales solo a los componentes más críticos.
- Analizar los modelos software y hardware donde las variables se discretizan y los rangos se reducen drásticamente.
- Analizar los modelos del sistema de forma jerárquica, de tal manera que se posibilite el *divide y vencerás*.
- Automatizar toda la Verificación que sea posible.

Pero cada dominio de aplicación requiere diferentes métodos de modelado y diferentes enfoques de prueba. Además, incluso dentro del dominio de una aplicación particular, las diferentes fases del ciclo de vida se pueden atender mejor desde diversas herramientas y técnicas. Por ejemplo, un demostrador de teoremas podría ser mejor para analizar la correctitud de la descripción de nivel RTL de un circuito de la Transformada de Fourier, mientras que los métodos algebraicos de

derivación podrían ser mejores para analizar la correctitud de los refinamientos de un diseño a nivel de compuertas.

A pesar de que la Verificación formal completa de un sistema complejo es poco práctica en este momento, los métodos formales se pueden aplicar a diversos aspectos o propiedades de estos sistemas, y comúnmente se aplican a la especificación detallada, al diseño y a la Verificación de sus partes críticas, como en la aviación, en los sistemas aeroespaciales y en sistemas de seguridad crítica, como el monitoreo de la frecuencia cardíaca.

2. PERSPECTIVA DE LOS MÉTODOS FORMALES

Por décadas los investigadores han explorado enfoques matemáticos en la síntesis, el análisis, la especificación y la prueba de programas informáticos, y Floyd [8] fijó los objetivos para esas exploraciones: 1) la semántica de los lenguajes de programación, y 2) la especificación y el razonamiento acerca de los programas individuales. Estos objetivos se convirtieron en la idea clave de las afirmaciones inductivas, porque definieron la semántica de los lenguajes y el significado de los programas mediante relaciones entre las pre-condiciones, las instrucciones del programa, y las post-condiciones. Desde entonces la posibilidad de la prueba mecánica de programas, o la generación heurística de programas, ha dado paso a diversos sistemas exploratorios y perspectivas teóricas.

Pero en ese camino surgieron dos obstáculos a la aplicación práctica: 1) la dificultad para capturar todo el contenido semántico de los lenguajes de programación y los entornos operativos, y 2) el desafío para expresar el propósito funcional y no-funcional de un programa en su contexto de uso. La investigación condujo entonces a diversos conceptos importantes: la definición formal de las características complejas del lenguaje y la identificación de características innecesarias y complejas; los lenguajes de especificación para tipos abstractos de datos, procesos concurrentes y máquinas abstractas; una teoría de la abstracción detrás de las estructuras del sistema jerárquico; lógicas mecanizables que permitan el razonamiento computacional acerca de las propiedades del programa, y teorías de dominios, como seguridad, relojes síncronos, microprocesadores y compilación.

Al descubrir las aplicaciones prácticas en estos dominios y al elaborar ejemplos a pequeña y mediana escala, la práctica tomó una ruta diferente. Se alcanzó y definió la Verificación a través de razonamientos con base en casos, es decir, en pruebas, con numerosos criterios y estrategias para la buena práctica de las mismas, sobre todo en cobertura funcional y estructural. Estas revisiones proporcionaron el principal medio de control intelectual: *la comprobación mental de las propiedades deseables de los sistemas en desarrollo, y la comunicación simultánea entre las partes interesadas*. Por otro lado, las metodologías heurísticas para el análisis y el diseño de los requisitos estructurados ofrecen una guía adicional a los sistemas que capturan la sabiduría convencional de una buena estructura, y proporcionan un medio de comunicación común.

Luego de esto los investigadores desarrollaron una base teórica para la prueba y los resultados, y aunque en su mayoría fueron negativos sugirieron diversas heurísticas para las pruebas, que se aproximaban más a un ideal en el que cada caso de prueba significaba algo con alguna posibilidad de revelar errores o demostrar una nueva evidencia de correctitud. Las metodologías heurísticas de la práctica no llamaron la atención de los investigadores, aunque los tipos abstractos de datos dieron lugar a lenguajes orientados por objetos y a métodos para agregar aún más estructuras y soporte al desarrollo del sistema heurístico. Los resultados teóricos, como la compresión de datos, la corrección de errores y algoritmos encriptados para el almacenamiento en redes y en discos, y

las estructuras de datos que permiten la representación y la búsqueda en bases de datos y el procesamiento de imágenes, también han desempeñado un importante papel en el desarrollo de sistemas. Además, las teorías y estrategias para la gestión de la computación distribuida y los datos son especialmente exigentes, tanto sobre recursos físicamente distribuidos como en sistemas de computación con multi-procesadores.

Pero, sin importar el enfoque técnico que se aplique en el desarrollo de software, la necesidad del procesamiento de información común surge de mantener coherencia entre, y la inteligibilidad de, una amasa entrelazada de documentos que expresan los puntos de vista de las partes interesadas, con cambios constantes en la estructura, además de que las partes interesadas también cambian a lo largo de los años de vida útil del sistema. Los entornos de programación han evolucionado para hacer frente a estas necesidades, y hoy se reflejan en editores estructurados, gestión de configuración, representaciones de base de datos, interfaces gráficas, la forma de coordinar el flujo de trabajo, y en las propuestas para interactuar con las partes interesadas. Especialmente importantes son los métodos y herramientas que se pueden utilizar más allá de su contexto en el proyecto, por ejemplo, componentes software, plantillas de documentos, guías de revisión y de errores, y datos de productividad.

Otra cuestión importante en la práctica ha sido la atención forzada sobre los procesos de desarrollo del sistema, es decir, la forma en que una organización gestiona y mejora su infraestructura, y los procedimientos específicos. Pero la forma basada en la lógica de los enfoques matemáticos, para describir el sistema, madura mediante el razonamiento estadístico acerca de los errores y el crecimiento de la fiabilidad en el tiempo, con el objetivo de introducir el control de calidad industrial y las prácticas de seguridad.

3. APLICACIÓN DE LOS MÉTODOS FORMALES

Los métodos formales se han investigado desde el siglo pasado y se encuentran trabajos como los de Redes Petri, que se remontan a la década de 1960. Estas investigaciones se realizaron en el ámbito de las Ciencias Computacionales, y generalmente su objetivo era elaborar técnicas que se pudieran utilizar en los procesos de desarrollo de software. De hecho, algunos de los primeros defensores sugirieron que era posible sustituir inmediatamente a los métodos informales y *ad-hoc* con la especificación, la Verificación y el análisis formal. Sin embargo, aunque existen casos de aplicaciones exitosas y algunos nichos de mercado donde su aplicación es rutinaria, todavía no se masifica su aplicación en el desarrollo de programas.

Una posible explicación es que los métodos formales no se integran fácilmente en las prácticas actuales de desarrollo, en parte porque requieren un importante cambio notacional. Además, todavía no compaginan adecuadamente con algunos paradigmas de aplicación, como los lenguajes de programación imperativa. Sin embargo, los investigadores siguen valorando los frutos alcanzados y, de hecho, el éxito de las recientes aplicaciones de los métodos formales en el desarrollo de sistemas sustenta su punto de vista. Entonces, ¿qué es lo que ofrecen los métodos formales y cuáles son sus beneficios?

1. Soportan de forma abstracta la descripción del comportamiento de los sistemas. Esto es importante, porque, aunque los lenguajes de programación reconocen claramente el comportamiento de los sistemas, no lo describen de forma descriptiva; es decir, un programa solo logra definir una única implementación. Pero gracias a los principios no-determinísticos, los métodos formales pueden soportar una descripción más abstracta, donde caracterizan la especificación como un conjunto de posibles implementaciones.

2. Las descripciones formales de los sistemas son susceptibles de análisis y manipulación formal. Por ejemplo, lo que demuestra el teorema de gran alcance y el modelo de herramientas de control, que se utilizan para determinar las descripciones de las propiedades del sistema que se pueden exhibir. Además, este análisis se puede realizar de manera automática, o por lo menos, semi-automática. Por otro lado, los principios básicos de los métodos formales a menudo son muy generales, por ejemplo, parte de la investigación en la teoría de concurrencia subyace en los conceptos básicos, particularmente en el proceso de cálculo, lo que hace que estos sistemas se vean como una colección de componentes que evolucionan, al mismo tiempo que interactúan mediante la realización de citas sincrónicas. Estos principios son completamente generales y de ninguna manera vinculan el dominio de aplicación de las Ciencias Computacionales.

A pesar de que la comunidad de investigación es pequeña y de que la acogida en los procesos formativos de pregrado y postgrado todavía es poca, no se puede perder de vista los beneficios de los métodos formales. De hecho, poder desarrollar la capacidad para realizar descripciones abstractas del comportamiento de los sistemas, para analizarlas formalmente con el objetivo de determinar sus propiedades emergentes, es un caso de éxito de la investigación en métodos formales. A continuación, se describe algunas razones que sustentan el hecho de que es conveniente ampliar su aplicación:

1. Una ventaja de aplicar especificación y Verificación formal en un entorno de desarrollo es que a menudo el problema no se plantea claramente. Esto se debe a que típicamente los métodos formales solo se utilizan para el modelado y el análisis, y no para generar una implementación en el sentido de las Ciencias Computacionales.
2. La abstracción puede ser importante en estas áreas exóticas de aplicación, porque a menudo las interpretaciones programadas son muy prescriptivas y solo es posible una descripción suelta del sistema objeto de investigación. Esto se debe, por ejemplo, a que no se dispone de una comprensión completa y determinística del comportamiento del sistema, o a que ciertos aspectos son tan complejos que no se pueden describir determinísticamente.
3. La formalización brinda la posibilidad de verificar la descripción resultante de diferentes formas: razonando acerca de la descripción y uso del teorema de la tecnología de pruebas, verificando el modelo para determinar automáticamente si tiene una propiedad particular, o hacer un análisis de resultados para determinar el comportamiento del sistema. Todas estas técnicas tienen una importancia relevante en áreas particulares de aplicación:
   - Descripción formal de juegos
   - Manuales de usuario
   - Formalización y Verificación de procedimientos y protocolos
   - Música formalizada
   - Representación formal de novelas y textos narrativos
   - Conducta personal/social de agentes humanos y virtuales
   - Temáticas de lenguaje natural
   - Modelado de usuario formal para la interacción Persona-Computador.
   - Modelado formal de sistemas biológicos
   - Modelos físicos formales

4. EL FUTURO DE LOS MÉTODOS FORMALES

En términos de aplicación en el siglo XXI los métodos formales están entrando en un período prometedor, y diferentes industrias, como la del hardware y software, la aeroespacial, y la

automotriz, ya los han adoptado. Como se ha demostrado convincentemente [14-16], la demanda por más seguridad en el software obliga a su utilización, y las herramientas formales se despliegan como una singularidad en navegadores y sistemas operativos. En Europa se ha desarrollado aplicaciones importantes como las tarjetas inteligentes para el metro de París, y un currículo dinámico que incluye la formación en métodos formales a nivel de pre y postgrado. También se aplican en las ciencias, como en los trabajos de Cardelli en biología de sistemas para el uso de *cuadernos de laboratorio formales*. En EE.UU., la NSA y NASA organizan un trabajo aplicado de los métodos formales en materia de seguridad y en los vuelos aeroespaciales.

Se espera que a medida que se incremente y difunda la aplicación de los métodos formales surja nuevos y fundamentales desafíos para su investigación. Uno de ellos podría ser el de las librerías matemáticas formales, que requieren una comprensión más profunda de cómo compartir los resultados entre los probadores. Por otro lado, se espera que estos desafíos impregnen y exciten a las Ciencias Computacionales en su totalidad. Además, una mirada seria desde los métodos formales a la comunidad de la *Web semántica* podría crear nuevos desafíos, y demostrar que es posible mover la computación a la esfera semántica.

También se podría pensar en incrementar la eficacia de las pruebas de software, lo que originaría un área importante de trabajo investigativo en Ciencias Computacionales. Puede que sea posible relacionar esa investigación con la que se lleva a cabo en las redes, la teoría computacional, o el aprendizaje autónomo de las máquinas, y las preocupaciones y desafíos de otras comunidades cercanas a los métodos formales, como las matemáticas computacionales, con lo que se podría vivir un período de alto impacto de la investigación en esta área.

La mayoría de las aplicaciones conocidas de los métodos formales busca automatizar completamente y controlar los modelos. Pero muchos de los proyectos que demuestran teoremas centrales se basan en la construcción de pruebas interactivas de teoremas complejos. Como trabajo siguiente se podría observar un compromiso más serio para usar ampliamente las aserciones y datos, tipo invariantes, en el código ordinario, lo que despertaría el interés y la realización de herramientas formales que les ayuden a los desarrolladores y diseñadores a comprender lo que están haciendo, y a exponer la estructura lógica subyacente de los algoritmos. Esta comprensión aceleraría el proceso de desarrollo y lo haría más confiable, porque podrá moverse a un nivel superior de abstracción.

5. CONCLUSIONES

Las fuerzas naturales que impulsan la aplicación de los métodos formales, las que ofrecen plataformas más potentes, y las extensiones de las herramientas estándar que vienen con cada aplicación nueva e importante, facilitaran que la investigación de los métodos formales genere aplicaciones diversas y creativas, y será posible hacer realidad más casos de éxito en esta área, como las expediciones NASA, donde podrían tener mayor impacto los avances en el conjunto básico de herramientas y aplicaciones.

Otra cuestión que se debe atender es cómo introducir los métodos formales en los procesos formativos, y cómo hacer que los estudiantes de pregrado y postgrado se interesen en ellos. Esta es un área en la que las necesidades son inmediatas, porque si desde ahora se trabaja pensando en el relevo generacional, la investigación en métodos formales podría ofrecer productos y resultados de mayor impacto. No se puede esperar hasta que los investigadores actuales ya no estén, porque no habrá quién capacite a los nuevos interesados.

Una oportunidad para lograrlo es mediante cooperación entre las universidades y la industria que trabaja en métodos formales, porque esto generaría modificaciones al currículo para que los estudiantes utilicen técnicas y herramientas formales, como la especificación formal, las aserciones, los modelos de control integrado, los avances en el refinamiento formal, y los protocolos para diseñar y aplicar pruebas. El logro efectivo de esta oportunidad supone un trabajo amplio, pero el esfuerzo será recompensado con beneficios formativos, y pondrá a los métodos formales al lado de las temáticas centrales de las Ciencias Computacionales y la ingeniería.

REFERENCIAS

[1] Meyer B. (1995). From Process to Product: Where is Software Headed? IEEE Computer 28(8), 17-23.

[2] Serna E. (2010). Formal Methods and Software Engineering. Revista Virtual Universidad Católica del Norte 30, 158-184.

[3] Saiedian H. (1996). An Invitation to Formal Methods. IEEE Computer 29(4), 16-17.

[4] Meyer B. (1997). The Next Software Breakthrough. IEEE Computer 30(7), 113-114.

[5] Craigen D. y Summerskill K. (1991). Formal Methods for Trustworthy Computer Systems. Report from FM89: A Workshop on the Assessment of Formal Methods for Trustworthy Computer. Springer.

[6] Knight J. et al. (1997). Why are formal methods not used more widely? En The Fourth NASA Langley Formal Methods Workshop. Hampton, Virginia.

[7] Serna E. (2014). Formal Methods in Context. En XIII Conferencia Iberoamericana en Sistemas, Cibernética e Informática CISCI. Orlando, USA.

[8] Floyd R. (1967). Assigning Meanings to Programs. Mathematical aspects of computer science 19, 19-32.

[9] Bloomfield R. (1986). The Application of Formal Methods to the Assessment of High Integrity Software. IEEE Transactions on Software Engineering 12(9), 988-993.

[10] Gries D. (1981). The Science of Programming. Springer.

[11] Hoare C.A.R. (1969). An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576-583.

[12] Dijkstra E.W. (1976). A Discipline of Programming. Prentice-Hall.

[13] Akers R. et al. (1990). Gypsy Verification Environment User's Manual. Technical Report 61. Computational Logic Inc.

[14] Craigen D. (1995). Formal Methods Technology Transfer: Impediments and Innovation. Lecture Notes in Computer Science 962, 328-332.

[15] Bowen J. y Hinchey M. (1995). Applications of Formal Methods. Prentice-Hall.

[16] Woodcock J. et al. (2009). Formal methods: Practice and experience. ACM Computing Surveys 41(4), 1-40.

CAPÍTULO III
Métodos formales e Ingeniería del Software[1]

Edgar Serna M.
Instituto Antioqueño de Investigación

Los métodos formales surgieron como puntos de vista analíticos con los que es posible verificar el desarrollo de sistemas mediante la lógica y las matemáticas, lo que aporta ventajas para mejorar la calidad de los programas y por tanto a la Ingeniería del Software. En este campo del conocimiento la especificación formal es una de las más importantes fases del ciclo de vida, labor que requiere cuidado ya que su función es garantizar que tanto el funcionamiento como el desempeño del programa sean correctos, bajo cualquier situación. En el futuro los métodos formales deberían estar presentes como principios esenciales en el desarrollo de software, ya que se convierten en la base para aplicar las técnicas de prueba y, dado su principio matemático, en potencialmente automatizables.

[1] Publicado en la Revista Virtual Universidad Católica del Norte 30 (pp. 158-184). 2010.

INTRODUCCIÓN

El concepto de los métodos formales involucra una serie de técnicas lógicas y matemáticas con las que es posible especificar, diseñar, implementar y verificar los sistemas de información [1]. Hasta hace pocos años el desarrollo industrial de sistemas, usando dichas técnicas, era considerado un complejo ejercicio teórico e inviable en problemas reales. Las notaciones *oscuras* tomadas de la lógica, sin las suficientes herramientas de soporte, no podían competir con los lenguajes de cuarta generación y los ambientes de desarrollo rápido de finales de siglo, sin embargo, el mundo industrial ha cambiado su actitud frente a los métodos formales. Por una parte, porque los lenguajes de especificación son cada vez más cercanos a los lenguajes de programación, las técnicas de desarrollo se adaptan mejor a los nuevos paradigmas, y las herramientas comerciales que soportan la calidad del software son métodos que se distribuyen comercialmente [2]. Además, porque a medida que los sistemas informáticos crecen en complejidad las pérdidas causadas por fallas son cada vez mayores. Cuando *corrección certificada* se traduce en dinero, los métodos formales atraen a la industria, ya que su aplicación ayuda a lograr los estándares de calidad que la sociedad exige.

La importancia de los métodos formales en la Ingeniería del Software se incrementó en el siglo XXI: se desarrollaron nuevos lenguajes y herramientas para especificar y modelar formalmente, y se diseñaron metodologías maduras para verificar y validar. Los modelos que se diseñan y construyen de esta forma, desde las fases iniciales del desarrollo de software, son esenciales para el éxito del futuro proyecto, ya que en la actual Ingeniería del Software constituyen la base que sustenta las subsiguientes fases del ciclo de vida, y porque los errores surgidos en ella tienen gran impacto en los costos del proyecto [3].

Desde hace varias décadas se utiliza técnicas de notación formal para modelar los requisitos, principalmente porque estas notaciones se pueden verificar *fácilmente* y porque, de cierta forma, son más *comprensibles* para el usuario final [4]. Además, el paradigma Orientado por Objetos en la programación parece ser el más utilizado en la industria, y una forma de incrementar la confiabilidad del software y, en general, de los sistemas, es utilizar los métodos formales en la ingeniería aplicada [5].

Pero, aunque los métodos formales tienen un amplio recorrido y su utilidad y eficiencia en desarrollos críticos están demostradas, todavía falta más trabajo para que la mayoría de ingenieros los conozcan y apliquen. En parte esta labor la deben realizar las universidades, incluyéndolos en sus contenidos académicos; los profesores, que se deben formar e investigar mejor en esta área del conocimiento; y los estudiantes, que deben lograr una formación más sólida en matemáticas y lógica [6]. Solo con un trabajo transdisciplinar se podrá lograr que la labor del ingeniero de software sea verdadera Ingeniería, y que el usuario final acepte los productos software como confiables.

Los propósitos de los métodos formales son: 1) sistematizar e introducir rigor en todas las fases de desarrollo de software, con lo que es posible evitar que se pasen por alto cuestiones críticas; 2) proporcionar un método estándar de trabajo a lo largo del proyecto; y 3) constituir una base de coherencia entre las muchas actividades relacionadas y, al contar con mecanismos de descripción precisos y no ambiguos, proporcionar el conocimiento necesario para realizarlas con éxito.

Los lenguajes de programación utilizados para desarrollar software facilitan la sintaxis y la semántica precisas para la fase de implementación, sin embargo, la precisión en las demás fases debe provenir de otras fuentes. Los métodos formales se inscriben en una amplia colección de

formalismos y abstracciones, cuyo objetivo es proveer un nivel de precisión comparable para las demás fases. Si bien esto incluye temáticas que actualmente están en desarrollo, algunas metodologías ya alcanzaron un nivel de madurez suficiente para que se utilicen ampliamente.

Existe una tendencia discutible a fusionar la Matemática Discreta y los métodos formales en la Ingeniería del Software. Efectivamente, en muchas temáticas de la primera se apoya la segunda, y no es deseable evitarlas cuando se trabaja con métodos formales, pero no es posible pretender que por el simple hecho de tomar algunos enfoques de la Matemática Discreta y emplearlos en aquella, se apliquen métodos formales. El principal objetivo de la Ingeniería del Software es desarrollar sistemas de alta calidad, y con los métodos formales, en conjunción con otras aéreas del conocimiento, se puede lograr.

1. QUÉ SON LOS MÉTODOS FORMALES

Aunque el término se utiliza ampliamente no parece existir una clara definición acerca de qué son en realidad [7]. En muchas ocasiones el término se emplea simplemente para indicar la utilización de un lenguaje de especificación formal, pero no incluye una descripción de cómo se utiliza o la extensión de su uso. Incluso el término *lenguaje de especificación formal* no es preciso, ya que no es claro si los lenguajes *específicamente* tienen como objetivo diseñar la implementación, en lugar de especificar el problema.

En este capítulo se utiliza el término *métodos formales* para describir cualquier enfoque que utilice un lenguaje de especificación formal, pero que describa su función en el proceso del desarrollo de software. Utilizar métodos formales necesariamente implica un proceso formal de refinamiento, razonamiento y prueba. El término *lenguaje de especificación formal* se utiliza aquí para referenciar un lenguaje en el que es posible especificar completamente la funcionalidad de todo o parte de un programa, de tal forma que sea susceptible de razonamiento formal, y que se fundamenta en una sólida base matemática, más en que si es lo suficientemente abstracto.

La base teórica de los lenguajes de especificación formal varía considerablemente, la más común es la de los métodos formales. Los lenguajes de este tipo más utilizados en la industria son The Vienna Development Method VDM [8] y Zed Z que se basa en la teoría de conjuntos y la lógica de predicados de primer orden de Zermelo Fraenkel [9]. VDM se utilizó primero en proyectos de gran tamaño, pero Z parece ganar popularidad recientemente. Aunque su sintaxis es diferente ambos lenguajes están estrechamente relacionados, y se basan principalmente en la teoría matemática de conjuntos [10]. Según Gödel [11] un sistema matemático formal es *un sistema de símbolos con sus respectivas reglas de uso*, todas recursivas. Este último requisito es importante, porque permite desarrollar programas para comprobar si un determinado conjunto de reglas se aplica correctamente [12]; sin embargo, garantizar que se puede lograr utilizando métodos formales tiene un precio, ya que para muchas aplicaciones resulta extremadamente costosos.

Las nociones fundamentales de especificación y Verificación formal de programas se desarrollaron en los años 70, entre otros por Hoare [13] y Dijkstra [14]: *el primer paso en la construcción de un sistema es determinar un modelo abstracto del problema que se va a resolver sobre el cual razonar*. Como no todos los aspectos del problema son relevantes en el sistema que se desarrolla, es necesario una observación cuidadosa para abstraer sus características más importantes, y utilizar un lenguaje formal para describir el modelo (especificación formal). En las Ciencias Computacionales los métodos formales adquieren un sentido más lineal, y se refieren específicamente al uso de una notación formal para representar modelos de sistemas y, en un sentido aún más estrecho, a la formalización de un método para desarrollar sistemas [15].

Métodos formales se refiere entonces al uso de técnicas de la Lógica y de la Matemática Discreta para especificar, diseñar, Verificar, desarrollar y Validar programas. La palabra *formal* se deriva de la Lógica Formal, ciencia que estudia el razonamiento desde el análisis formal, de acuerdo con su validez o no-validez, y omite el contenido empírico del razonamiento para considerar solo la forma (estructura sin materia). Los métodos formales más rigurosos aplican estas técnicas para comprobar los argumentos utilizados para justificar los requisitos, u otros aspectos de diseño e implementación de un sistema complejo.

En la lógica formal como en los métodos formales el objetivo es el mismo: *reducir la dependencia de la intuición y el juicio humanos para evaluar argumentos* [16]. Los métodos menos rigurosos enfatizan en la formalización y renuncian a la computación, definición que implica un amplio espectro de técnicas, y una gama igualmente amplia de estrategias. En determinados proyectos, la interacción de las técnicas y estrategias de muchos métodos formales se limita por el papel que interpretan y por los recursos disponibles para su aplicación [17].

2. LOS MÉTODOS FORMALES EN LA INGENIERÍA DEL SOFTWARE

En la Ingeniería del Software un método formal es un proceso que se aplica para desarrollar programas, y que explota el poder de la notación y de las pruebas matemáticas. Además, los requisitos, la especificación y el sistema completo deben validarse con las necesidades del mundo real [5]. En la Ingeniería del Software los métodos formales se utilizan para:

1. *Las políticas de los requisitos*. En un sistema seguro se convierten en las principales propiedades de seguridad que éste debe conservar, es decir, el modelo de políticas de seguridad formal, como confidencialidad o integridad de datos.
2. *La especificación*. Es una descripción matemática basada en el comportamiento del sistema, que utiliza tablas de estado o lógica matemática. No describe normalmente al software de bajo nivel, pero sí su respuesta a eventos y entradas, de tal forma que es posible establecer sus propiedades críticas.
3. *Las pruebas de correspondencia entre la especificación y los requisitos*. Es necesario demostrar que el sistema, tal como se describe en la especificación, establece y conserva las propiedades de las políticas de los requisitos. Si están en notación formal se puede diseñar pruebas rigurosas manuales o automáticas.
4. *Las pruebas de correspondencia entre el código fuente y la especificación*. Aunque muchas técnicas formales se crearon inicialmente para probar la correctitud del código, pocas veces se logra debido al tiempo y costo implicados, pero pueden aplicarse a los componentes críticos del sistema.
5. *Pruebas de correspondencia entre el código máquina y el código fuente*. Este tipo de pruebas raramente se aplica debido al costo y a la alta confiabilidad de los compiladores modernos.

Por lo tanto, los métodos formales en la Ingeniería del Software son técnicas matemáticamente rigurosas que se utilizan para describir las propiedades del sistema, y proporcionan marcos de referencia para especificarlo, desarrollarlo y verificarlo de forma sistemática, en lugar de hacerlo *ad hoc* [18]. Un método es formal si posee una base matemática estable, normalmente soportada por un lenguaje de especificación formal, que permita definir, de manera precisa, nociones como consistencia y completitud y, aún más relevante, la especificación, la implementación y la correctitud. Al utilizar notaciones y lenguajes formales es posible estructurar claramente los requisitos del sistema y generar las especificaciones que permitan definir su comportamiento de acuerdo con el *qué debe hacer*, y no con el *cómo lo hace* [19].

Matemáticamente rigurosas significa que las especificaciones en los métodos formales son declaraciones gramaticalmente correctas en una lógica matemática, y que la verificación formal es una deducción rigurosa en ella. Es decir, que cada paso se deriva de una regla de inferencia que se puede comprobar mediante un proceso automático [20]. Los métodos formales en la Ingeniería del Software difieren en la forma como se aplican, y en los tiempos necesarios para cada fase del ciclo de vida; su estructuración y utilización requiere mayor tiempo y trabajo para desarrollar la especificación, pero su utilización garantiza, más *formalmente*, que la construcción de los diseños es correcta.

2.1 Fundamentos matemáticos de los métodos formales

La matemática es una ciencia de la cual se puede aprovechar muchas de sus propiedades para el desarrollo de los grandes y complejos sistemas actuales, en los que, idealmente, los ingenieros deberían estructurar su ciclo de vida de la misma forma que un matemático se dedica a la matemática aplicada: *presentar la especificación matemática del sistema y elaborar una solución con base a una arquitectura de software que haga posible su implementación* [21]. Aunque la especificación matemática de sistemas no es tan concisa como las expresiones matemáticas simples, dado que los sistemas software son mucho más complejos, no sería realista pensar que es posible especificarlos de la misma forma matemática. En todo caso, la ventaja de utilizar matemáticas en la Ingeniería del Software radica en que *suaviza* la transición entre sus actividades, ya que es posible expresar la especificación funcional, el diseño y el código de un programa mediante notaciones. Esto se logra gracias a que la propiedad fundamental de la matemática es la abstracción, una excelente herramienta para modelar debido a que es exacta, y a que ofrece pocas probabilidades de ambigüedad, lo que permite verificar matemáticamente las especificaciones y buscar contradicciones e incompletitudes.

Con las matemáticas es posible representar de forma organizada los niveles de abstracción en la especificación del sistema; es una buena herramienta para modelar, ya que facilita el diseño del esquema principal de la especificación; permite a analistas e ingenieros verificar la funcionalidad de esa especificación y a los diseñadores ver los detalles suficientes para llevar a cabo su tarea desde las propiedades del modelo. También proporciona un nivel elevado de Verificación, ya que para probar que el diseño se ajusta a la especificación y que el código refleja exactamente el diseño, se puede utilizar una demostración matemática [22].

Como se expresó antes, el término formal se caracteriza por utilizar una serie de categorías de modelos o formalismos matemáticos, que no solo incluyen los fundamentos matemáticos, sino también a los procesos para la producción de software. Entre estos fundamentos básicos se cuenta nociones como lenguajes formales, sintaxis, semántica, modelos, gramática, teorías, especificación, Verificación, Validación y pruebas; temáticas a las que deberían tener acceso los ingenieros de software para contar con bases sólidas al momento de seleccionar un determinado formalismo [23-25], desde los cuales es posible, de forma más acertada, tener los primeros acercamientos a conceptos como requisitos, especificación formal e informal, validación, validez, completitud, correctitud y nociones sobre pruebas del software. Los formalismos matemáticos de los métodos formales son:

- *Lógica de primer orden y teoría de conjuntos* [26]. Se utilizan para especificar el sistema mediante estados y operaciones, de tal forma que los datos y sus relaciones se describan detalladamente, sus propiedades se expresen en lógica de primer orden, y la semántica del lenguaje tenga como base la teoría de conjuntos. En el diseño y la implementación del sistema los elementos, descritos matemáticamente, se pueden modificar, pero siempre conservarán

las características con las que se especificaron inicialmente. Mediante diversos sistemas de deducción en esta lógica se establece la definición del lenguaje empleado (lógica de predicados de primer orden) y de la prueba. Tanto las reglas de inferencia, el diseño de las pruebas, como el estudio de las teorías de igualdad e inducción, representan aspectos claves de este formalismo [25, 27].

- *Algebraicos y de especificación ecuacional* [28]. Mediante fórmulas ecuacionales describen las estructuras de datos de forma abstracta para establecer el nombre de los conjuntos de datos, sus funciones básicas y propiedades, en las que no existe el concepto de estado modificable en el tiempo. Tanto el cálculo ecuacional (pruebas mediante ecuaciones), como los sistemas de reescritura, constituyen el soporte para realizar las deducciones necesarias [29]. También se utilizan para especificar sistemas de información a gran escala, tarea en la que es necesario estudiar los mecanismos de extensión, de parametrización y de composición de las especificaciones [28].

- *Redes de Petri* [30-35]. Establecen el concepto de estado del sistema mediante lugares que pueden contener marcas, y hacen uso de un conjunto de transiciones, con pre y post-condiciones, para describir cómo evoluciona el sistema y cómo produce marcas en los puntos de la red. Utilizan características semánticas de entrelazado o de base en la concurrencia real: en la primera, los procesos paralelos no deben ejecutar las instrucciones al mismo tiempo, mientras que en la segunda existe esa posibilidad. Estas caracterizaciones son necesarias para representar conceptos como estado y transición en los sistemas.

- *Lógica temporal* [32]. Se utiliza para describir los sistemas concurrentes y reactivos, y posee una amplia noción de tiempo y estado. Sus especificaciones describen las secuencias válidas de estados, incluyendo los concurrentes, en un sistema específico. Para aplicar este formalismo es necesario establecer inicialmente una clasificación de los diferentes sistemas de lógica temporal, de acuerdo con los criterios de proposicionalidad vs. primer orden, tiempo lineal vs. tiempo ramificado, de evaluación instantánea o por intervalos, y tiempo discreto vs. tiempo continuo. Una vez que se establece la formalización del tiempo es posible estudiar la aplicación del formalismo.

Los métodos formales proporcionan un medio para examinar simbólica y completamente el diseño digital, bien sea hardware o software, y establecer las propiedades de correctitud o seguridad. Sin embargo, en la práctica raramente se logra, excepto en los componentes de seguridad de sistemas críticos, debido a la enorme complejidad de los sistemas reales. En varios proyectos se ha experimentado con los métodos formales para lograr el desarrollo de sistemas reales con calidad:

- Para automatizar la verificación tanto como sea posible [36].
- En requisitos y diseños de alto nivel, en los que la mayor parte de los detalles se abstraen de diversas formas [37].
- Solo a los componentes más críticos [38].
- Para analizar modelos software y hardware en los que las variables se discretizan y los rangos se reducen drásticamente [39].
- En el análisis de los modelos de sistemas de manera jerárquica, de tal forma que se aplique el concepto: *divide y vencerás* [40].

Aunque el uso de la lógica matemática sea un tema de unificación en los métodos formales, no es posible indicar que un método sea mejor que otro; cada dominio de aplicación requiere métodos

de modelado diferentes, lo mismo que el acercamiento a las pruebas. Además, en cada dominio en particular las diferentes fases del ciclo de vida se pueden lograr mejor con la aplicación combinada de diferentes técnicas y herramientas formales [41].

El siguiente ejemplo ilustra la utilización de los métodos formales en la Ingeniería del Software. Se trata de especificar y de utilizar, de forma diferente a la tradicional, el lenguaje matemático para representar los requisitos de los sistemas. El ejemplo muestra la especificación de las funciones de inserción y borrado en una estructura pila.

Tabla 1. Especificación en lógica formal de las funciones de insertar y borrar en una estructura Pila

▪ **L.push**(e:stelement): Inserta *e* después del último elemento insertado
▪ **PRE**: ∃L ∧ L ≠ {NULL} ∧ L = {PRF} ∧ tamaño(L) = n ∧ ¬existe(L,clave(e))
▪ **POST**: L = {PRF,e} ∧ tamaño(L) = n + 1
▪ **PRE**: ∃L ∧ L = {NULL}
▪ **POST**: L = {e} ∧ n = 1
▪ **L.pop**(): borra el elemento recientemente insertado
▪ **PRE**: ∃L ∧ L = {PRF, e} ∧ tamaño(L) = n
▪ **POST**: L = {PRF} ∧ tamaño(L) = n – 1

Nótese que, en lugar de utilizar el lenguaje natural para especificar las funciones, se recurre a la notación Z de la lógica formal para hacerlo. Haciéndolo de esta forma no se tiene ambigüedades y es posible automatizarlas.

3. UTILIDAD DE LOS MÉTODOS FORMALES PARA LA INGENIERÍA DEL SOFTWARE

La utilidad de los métodos formales en la Ingeniería del Software es un tema que se debate desde hace varias décadas. Recientemente, con el surgimiento de la Ingeniería de Conocimiento en la Sociedad de la Información y la aplicación de los métodos formales en los procesos industriales, nuevamente surge el debate. A continuación, se detalla algunas de las ventajas de utilizarlos.

3.1 Especificación

Uno de los problemas más ampliamente reconocido en el desarrollo de software es la dificultad para especificar claramente el comportamiento que se espera del software, problema que se agudiza con el desarrollo basado en componentes, ya que el ingeniero tiene solo una descripción textual de los requisitos, procedimientos, entradas permitidas y salidas esperadas. No es fácil asegurar que este tipo de software sea seguro, no solo por su tamaño y complejidad, sino porque el código fuente no suele estar disponible para los componentes que se adquieren. Una forma de ofrecer *garantía rigurosa* del producto es definir con precisión el comportamiento esperado del software [42].

La especificación formal proporciona mayor precisión en el desarrollo de software, y los métodos formales brindan las herramientas que pueden incrementar la garantía buscada. Desarrollar formalmente una especificación requiere conocimiento detallado y preciso del sistema, lo que ayuda a exponer errores y omisiones, por lo que la mayor ventaja de los métodos formales se da en el desarrollo de la especificación [43]. En la formalización de la descripción del sistema se detecta ambigüedades y omisiones, y una especificación formal puede mejorar la comunicación entre ingenieros y clientes.

Los métodos formales se desarrollaron principalmente para permitir un mejor razonamiento acerca de los sistemas y el software. Luego de diseñar una especificación formal se puede analizar,

manipular y razonar sobre ella, de la misma forma que sobre cualquiera expresión matemática. Una diferencia significativa entre una especificación formal de software y una expresión matemática de álgebra o cálculo, es que típicamente es mucho más grande, a menudo cientos o miles de líneas. Para tratar con el tamaño y la complejidad de estas expresiones se desarrolla herramientas software que se pueden agrupar en dos categorías: 1) probadoras de teoremas, y 2) verificadoras de modelos [44].

Las primeras ayudan al usuario a diseñar pruebas, generalmente para demostrar que las especificaciones cumplen con las propiedades deseadas. Para usarlas es necesario que el ingeniero tenga cierto grado de habilidad y conocimiento, pero pueden manipular especificaciones muy grandes con propiedades complejas. Los desarrollos recientes en métodos formales introdujeron las verificadoras de modelos, que exploran, hasta cierto punto, todas las posibles ejecuciones del programa especificado.

Mediante la exploración de todas las posibles ejecuciones pueden verificar si cumple con una propiedad específica, o producen contraejemplos en los que la propiedad no se cumple. Aunque estas herramientas pueden ser completamente automáticas, no pueden resolver problemas tan grandes o variados como las probadoras de teoremas. Las herramientas más sofisticadas combinan aspectos de ambas y aplican las verificadoras a algunas partes de la especificación, pero confían en el usuario para probar las propiedades complicadas.

3.2 Verificación

Para asegurar la calidad de los sistemas es necesario probarlos, y para asegurar que se desarrollan pruebas rigurosas se requiere una precisa y completa descripción de sus funciones, incluso cuando en la especificación se utilicen métodos formales. Una de las aplicaciones más interesantes de éstos es el desarrollo de herramientas que pueden generar casos de prueba completos desde la especificación formal.

Aunque un amplio número de herramientas para automatizar pruebas se encuentra disponible en el mercado, la mayoría automatiza solo sus aspectos más simples: generan los datos de prueba, ingresan esos datos al sistema y reportan resultados. Definir la respuesta correcta del sistema, para un determinado conjunto de datos de entrada, es una tarea ardua que la mayoría no puede lograr cuando el comportamiento del mismo se especifica en lenguaje natural. Debido a que la respuesta esperada del sistema se puede determinar únicamente mediante la lectura de la especificación, los ingenieros esperan que a las pruebas automatizadas se les adicione este faltante y crítico componente [45]. La ventaja de las herramientas que generan pruebas con base en los métodos formales es que la especificación formal describe matemáticamente el comportamiento del sistema, desde la que se puede generar la respuesta a un dato de entrada en particular, es decir, la herramienta puede generar casos de prueba completos.

Las técnicas de verificación formal dependen de especificaciones matemáticamente precisas y, desde un punto de vista costo-beneficio, generar pruebas desde la especificación puede ser uno de los usos más productivos de los métodos formales. En un desarrollo típico de software comercial, aproximadamente la mitad del tiempo del equipo de trabajo se invierte en esfuerzo para desarrollar pruebas, e *incluso con este nivel de esfuerzo solo se elimina los errores más evidentes* [46]. Algunas mediciones empíricas demuestran que las pruebas, generadas con herramientas automatizadas, ofrecen una cobertura tan buena o mejor que la alcanzada por las manuales, por lo que los ingenieros pueden elegir entre producir más pruebas en el mismo tiempo, o reducir el número de horas necesarias para hacerlas [47].

3.3 Validación

Mientras que la Verificación se puede realizar semi-automáticamente y las pruebas mecánicamente, la Validación es un problema diferente. Una diferencia específica es que la primera responde a *¿se está construyendo el producto correctamente*? y la segunda a *¿se está construyendo el producto correcto?* [48]. En otras palabras, la Verificación es el conjunto de actividades que aseguran que el software implementa correctamente una función específica, y la Validación es un conjunto de actividades diferentes que aseguran que el software construido corresponde con los requisitos del cliente [49].

Desde el conjunto de requisitos es posible verificar, formal o informalmente, si el sistema implementa los requisitos, sin embargo, la Validación es necesariamente un proceso informal. Solo el juicio humano puede determinar si el sistema que se especificó y desarrolló es el adecuado para el trabajo. A pesar de la necesidad de utilizar este juicio en el proceso de validación, los métodos formales tienen su lugar, especialmente en grandes y complejas aplicaciones, como en modelado y simulación. Una de sus aplicaciones más prometedoras es en el modelado de requisitos, ya que, al diseñarlos formalmente, el teorema provisto en la herramienta de prueba se puede utilizar para explorar sus propiedades, y a menudo detectar los conflictos entre ellos. Este método no sustituye al juicio humano, pero puede ayudar a determinar si se especificó el *sistema correcto*, por lo que es más fácil determinar si se mantienen las propiedades deseadas [15].

Una diferencia significativa entre validar sistemas de modelado y simulación, y los de control o cálculo, es que los primeros tienen dos tipos de requisitos de validación: 1) deben modelar y predecir el comportamiento de alguna entidad del mundo real, problema que se conoce como *validación operacional*, y 2) deben *validar el modelo conceptual* para asegurar que la hipótesis en la que se sustenta es correcta, y que su lógica y estructura son adecuadas para el modelo que se propone [50]. Debido a que el modelo conceptual describe lo que debe representar la simulación, es necesario incluir supuestos acerca del sistema, su entorno, las ecuaciones, los algoritmos, los datos y de las relaciones entre las entidades del modelo. Aunque los algoritmos y las ecuaciones son declaraciones necesariamente formales, los supuestos y las relaciones se describen normalmente en lenguaje natural, lo que introduce potenciales ambigüedades e incomprensiones entre ingenieros y usuarios.

Una tendencia en los métodos formales, conocida como *métodos formales ligeros* [51], demuestra tener potencial para detectar errores importantes en la declaración de requisitos, sin el costo de una verificación diseñada formalmente. La premisa básica de este enfoque es el uso de técnicas formales en el análisis de los supuestos, las relaciones y las propiedades de los requisitos, indicadas en su declaración o en el modelo conceptual. Se puede aplicar a especificaciones parciales o a un segmento de la especificación completa, proceso que se realiza en tres fases: 1) reafirmar los requisitos y el modelo conceptual en una notación formal, o semi-formal, típicamente en una tabla de descripción de estados; 2) identificar y corregir las ambigüedades, conflictos e inconsistencias; y 3) utilizar un verificador de modelos o un probador de teoremas para estudiar el comportamiento del sistema, demostrar sus propiedades y graficar su comportamiento. Los Ingenieros y usuarios pueden utilizar estos resultados para mejorar el modelo conceptual [52].

Un aspecto particularmente interesante de este enfoque es que se ha utilizado para modelar y analizar el comportamiento del software, del hardware y de las acciones humanas en los sistemas [53]. Agerholm y Larsen [54] describen su aplicación en un sistema de actividad extra-vehicular de NASA, y Lutz [55] refiere la validación de requisitos de los monitores de errores a bordo de una

nave espacial. Un detalle importante de este proyecto es que los ingenieros utilizan el modelo de requisitos para un segundo proyecto, que se desarrolla a partir del primero, como una construcción en serie. Janssen et al. [56] describen la aplicación de un verificador de modelos para analizar procesos de negocios automatizados, como el procesamiento de reclamaciones.

4. PROSPECTIVA DE LOS MÉTODOS FORMALES

La industria del software tiene una larga y bien ganada reputación de no cumplir sus promesas y, a pesar de más de 60 años de progreso, continúa años décadas por debajo de la madurez necesaria que requiere para satisfacer las necesidades de la Sociedad del Conocimiento. A finales del Siglo XX Goguen [57] citaba algunas estimaciones de los gastos que generan los fracasos del desarrollo de software, que calculó en 81 mil millones de dólares para 1995 y en 100 mil millones para 1996. Posteriormente [58], llamó la atención sobre la cancelación del contrato de 8 mil millones de dólares a IBM por la The Federal Aviation Administration FAA, para el diseño de un sistema de control aéreo para toda la nación; del contrato del United States Department of Defense DOD a la misma IBM, por $2 mil millones para modernizar su sistema de información; del fallo del software para la entrega en tiempo real de datos en las Olimpiadas de 1996; y del año y medio de retraso en el sistema para el manejo automatizado de equipaje en el aeropuerto de Denver para United Airlines, con un costo de $1,1 millones diarios.

Por su parte, Neumann [59] revela que estos problemas no son en absoluto nuevos, aunque parece que se incrementan; incluso señala que algunos de ellos ya provocaron muertes de personas, por ejemplo, la sobredosis de radiación en un sistema de terapia a mediados de los 80 [60]. Es claro que con la tecnología actual aún no es posible asegurar el éxito de los proyectos software, y que para proyectos grandes y complejos el enfoque *ad hoc* ha demostrado ser insuficiente.

El asunto es la falta de formalización en los puntos clave de la Ingeniería del Software la hacen sensible a problemas que son inevitables en actividades altamente técnicas y detalladas como la creación de software. Las buenas prácticas en ingeniería deben aplicarse en todo el proceso del desarrollo de sistemas, pero, aunque el desarrollo tecnológico aporta mucho material para alcanzarlo, todavía no se logra este objetivo. Incrementar la precisión y el control riguroso es esencial, y es el principal objetivo de los métodos formales, que utilizan esencialmente formalismos lógicos buscando mejorar el software y el hardware, en áreas como confiabilidad, seguridad, productividad y reutilización. Los ejes principales de su accionar son la verificación del código y el diseño, así como la generación de programas y casos de prueba desde las especificaciones.

Los métodos formales deberían estar presentes como principios esenciales de las técnicas de prueba, al punto que Gaudel [] (1995) lo estableció como un tema importante de investigación; Hoare [62] describió el uso de aserciones formales no para probar el programa, sino para diseñar las pruebas; y Hierons et al. [63] desarrollaron una investigación en aspectos formales de las pruebas. Los métodos formales se utilizan en el mantenimiento del software [64] y en su evolución [65], y tal vez su más amplia aplicación sea en el mantenimiento de código heredado [66].

Los métodos formales son un área de investigación muy activa y se espera que cada día se incremente las colaboraciones. Existe varias revistas especializadas, como Formal Aspects of Computing y Formal Methods in System Design, que hacen hincapié en sus aplicaciones prácticas, así como en la teoría. Conferencias como NASA Formal Methods Symposium y Computer-Aided Verification están dedicadas al tema, y tienen procesos de selección de aportes muy competitivos.

Otras conferencias importantes son: Principles of Programming Languages, Logic in Computer Science y Conference on Automated Deduction. Existen talleres y congresos especializados más pequeños, algunos enfocados en herramientas y técnicas como la ABZ, que cubre las notaciones Alloy, ASM, B y Z, y talleres de refinamiento. Unos debaten cuestiones teóricas específicas, como Integrated Formal Methods, y otros cubren áreas de aplicación, como Formal Methods for Open Object-based Distributed Systems y Formal Methods for Human-Computer Interaction. The Computer Science Bibliography contiene referencias a innumerables artículos, indexados por metadatos, muchos acerca de los métodos formales, especificación, verificación y validación, lo que presupone que para esta área la producción seguirá en incremento.

Se espera que la investigación, y de acuerdo con las proyecciones mostradas, demuestre un notable incremento en resultados, desarrollo y aplicación; y entonces será posible una mayor aceptación en la industria, mayor participación de las facultades de ingeniería, y la academia en general, y más trabajo práctico y experimental alrededor de los métodos formales. Con esto se podrá concretar el apoyo decidido a la visión de Hoare [67, 68] de *un mundo en el que los productos software siempre serán fiables, y que la labor de los ingenieros de software sea realmente Ingeniería*.

5. CONCLUSIONES

Los métodos formales son técnicas matemáticas, a menudo soportadas por herramientas, para el desarrollo de sistemas software y hardware. Su rigor matemático les permite a los ingenieros analizar y verificar sus modelos en cualquier parte del ciclo de vida del desarrollo, y dado que la fase más importante en estos procesos es la Ingeniería de Requisitos, son útiles para elicitarlos, articularlos, representarlos y especificarlos [69]. Sus herramientas proporcionan el soporte automatizado para la integridad, trazabilidad, verificabilidad, reutilización, y para apoyar la evolución de los requisitos, los puntos de vista diversos y la gestión de las inconsistencias [70].

Los métodos formales se utilizan en la especificación de software y se emplean para desarrollar una declaración precisa de lo que el software tiene que hacer, evitando al mismo tiempo las restricciones del cómo se quiere lograr. La especificación es un contrato técnico entre el ingeniero y el cliente, que proporciona un entendimiento común de la finalidad del software; el cliente la utiliza para orientar la aplicación del software y el ingeniero para guiar su construcción. Una especificación compleja se puede descomponer en sub-especificaciones, que describen un sub-componente del sistema que se puede delegar a otros ingenieros (diseño por contrato) [71].

Los sistemas software complejos requieren una cuidadosa organización de la estructura arquitectónica de sus componentes, un modelo del sistema que suprima detalles de la implementación, que permita al arquitecto concentrarse en los análisis y decisiones más importantes para estructurar el sistema y satisfacer sus requisitos [72, 73]. Darwin [74] y Wright [75] son ejemplos de lenguajes de descripción arquitectónica basados en la formalización del comportamiento abstracto de los componentes y conectores arquitectónicos.

Los métodos formales se utilizan en el diseño de software, donde el refinamiento de datos incluye la especificación de máquinas de estado, funciones de abstracción y pruebas de simulación [76], y cumplen un rol protagónico en métodos como VDM y Z, y en el cálculo de refinamiento de programas [77].

En la fase de implementación los métodos formales se utilizan para verificar el código, ya que toda especificación tiene explícito un teorema de correctitud con el que, si se cumplen ciertas

condiciones, el programa deberá conseguir el resultado descrito en la documentación. La verificación del código es un intento por demostrar este teorema, o al menos de encontrar por qué falla. El método de verificación por aserción inductiva de programas fue desarrollado por Floyd [78] y Hoare [13], y consiste en anotar el programa con aserciones matemáticas, o sea las relaciones entre las variables y los valores iniciales de entrada.

Más de cuatro décadas de investigación y experimentación demostraron que los métodos formales son, hasta el momento, el mejor medio práctico para exponer la ausencia de comportamientos no deseados en los programas, una propiedad esencial en los sistemas críticos. Los modelos para probar calidad industrial y los probadores de teoremas avanzados permiten, de forma automática o semi-automática, realizar análisis complejos de las especificaciones formales, por lo que estas herramientas son atractivas para uso comercial.

La capacidad para generar casos de prueba completos desde la especificación formal representa un ahorro sustancial a pesar del costo de su desarrollo. La experiencia demuestra que las técnicas formales se pueden aplicar productivamente, incluso en las pruebas más completas. El proceso para desarrollar una especificación es la fase más importante de la verificación formal, y el enfoque de los *métodos formales ligeros* permite analizar formalmente las especificaciones parciales, y la definición temprana de requisitos.

REFERENCIAS

[1] Monin J. (2003). Understanding formal methods. Springer.
[2] Sobel K. y Clarkson R. (2002). Formal methods application: An empirical tale of software development. IEEE Transactions on software engineering 28 (3), 308-320.
[3] Perry W. (2006). Effective methods for software testing. Wiley.
[4] Felleisen M. et al. (2001). How to design programas: An Introduction to Programming and Computing. The MIT Press.
[5] Kuhn D. et al. (2002). Cost effective use of formal methods in verification and validation. En Workshop on Modeling and Simulation Verification and Validation for the 21st Century. Laurel, USA.
[6] Dijkstra E. W. (1989). On the cruelty of really teaching computing science. Communications of the ACM 32(12), 1398-1404.
[7] Hehner E. (2006). A practical theory of programming. Springer.
[8] Jones C. (1990). Systematic software development using VDM. Prentice-Hall.
[9] Spivey J. (1992). The Z Notation: A reference manual. Prentice-Hall.
[10] Hayes I. et al. (1994). Understanding the Differences between VDM and Z. ACM Software Engineering Notes 19(3), 75-81.
[11] Gödel K. (1934). On undecidable propositions of formal mathematical systems. Collected works I, 346-369.
[12] Zeugmann T. y Zilles S. (2008). Learning recursive functions: A survey. Theoretical Computer Science 397(1-3), 4-56.
[13] Hoare C.A.R. (1969). An axiomatic basis for computer programming. Communications of the ACM 12(10), 576-580.
[14] Dijkstra E.W. (1976). A discipline of programming. Prentice-Hall.
[15] Flynn S. y Hamlet D. (2006). On formal specification of software components and systems. Electronic Notes in Theoretical Computer Science 161, 91-107.
[16] Kiniry J. y Zimmerman D. (2008). Secret Ninja formal methods. En 15th international symposium on Formal Methods. Turku, Finland.
[17] García J. et al. (2006). TOP 10 de factores que obstaculizan la mejora de los procesos de verificación y validación en organizaciones intensivas en software. Revista Española de Innovación, Calidad e Ingeniería del Software 2(2), 18-28.
[18] Jones C. et al. (2006). Verified software: A Grand Challenge. Computer 39(4), 93-95.

[19] Bolstad M. (2004). Design by Contract: A simple technique for improving the quality of software. En The 2004 Users Group Conference. Williamsburg, USA.
[20] Liu Z. y Venkatesh R. (2005). Methods and tools for formal software engineering. Verified Software: Theories, Tools, Experiments. En First IFIP TC 2/WG 2.3 Conference. Zurich, Switzerland.
[21] Drechsler R. (2004). Advanced formal verification. Kluwer Academic Publishers.
[22] Langari Z. y Pidduck A. (2005). Quality, cleanroom and formal methods. En The third workshop on Software quality. St. Louis, USA.
[23] Wirsing M. (1991). Algebraic Specification. En Handbook of Theoretical Computer Science, Formal Models and Semantics (pp. 675-788). Pearson.
[24] Lewis H. y Papadimitriou C. (1997). Elements of the Theory of Computation. Prentice-Hall.
[25] Mendelson E. (1997). Introduction to Mathematical Logic. Chapman & Hall/CRC.
[26] Manna Z. y Waldinger R. (1985). The Logical Basis for Computer Programming, Volume I: Deductive Reasoning. Addison-Wesley.
[27] Milner R. (1990). Operational and algebraic semantics of concurrent processes. En Handbook of theoretical computer science, Volume B: Formal models and semantics (pp. 1201-1242). McGraw-Hill.
[28] Ehrig H. y Mahr B. (1990). Fundamental of Algebraic Specification II: Specifications and constraints. Springer.
[29] Duffy D. (1991). Principles of Automated Theorem Proving. John Wiley & Sons.
[30] Murata T. (1989). Petri nets: Properties, analysis and applications. Proceedings of IEEE 77(4), 541-580.
[31] Reising W. (1991). Petri nets and algebraic specifications. Theoretical Computer Science, 80(1), 1-34.
[32] Manna Z. y Pnueli A. (1992). The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer.
[33] Bause F. y Kritzinger P. (2002). Stochastic Petri Nets. Friedrich Vieweg & Sohn Verlag.
[34] Desel J. y Esparza J. (2005). Free Choice Petri Nets - Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.
[35] Juhás G. et al. (2007). Semantics of Petri nets: a comparison. En 39th conference on Winter simulation. Washington, USA.
[36] Wunram J. (1990). A Strategy for Identification and Development of Safety Critical Software Embedded in Complex Space Systems. En 41st International Astronautical Congress. Dresden, Germany.
[37] Rushby J. (1993). Formal Methods and Digital Systems Validation for Airborne Systems. SRI-CSL-93-07. Technical Report: cr4551. NASA Langley Technical Report Server.
[38] Lutz R. y Ampo Y. (1994). Experience report: Using formal methods for requirements analysis of critical spacecraft software. En 19th Annual Software Engineering Workshop. Greenbelt, USA.
[39] McDermid J. et al. (1995). Experience with the Application of HAZOP to Computer-Based Systems. En 10th Annual Conference on Computer Assurance. Gaithersburg, USA.
[40] Zhang Z. et al. (1999). Perspective-based Usability Inspection: An Empirical Validation of Efficacy. Empirical Software Engineering 4(1), 43-69.
[41] Pressman R. (2004). Software Engineering: A Practitioner's Approach. McGraw-Hill.
[42] NASA. (1995). Formal Methods Specification and Verification. Guidebook for Software and Computer Systems. Volume I: Planning and Technology Insertion. NASA-GB-002-95.
[43] Clarke E. y Wing J. (1996). Formal Methods: State of the Art and Future Directions. En Special ACM 50th-anniversary issue: Strategic directions in computing research (pp. 626-643). ACM.
[44] Anderson R. et al. (1998). Model checking large software specifications. IEEE Transactions on Software Engineering 24(7), 498-520.
[45] Dan L. y Aichernig B. (2002). Automatic Test Case Generation for RAISE. Report 273. International Institute for Software Technology. Macau, China.
[46] Saiedian H. y Hinchey M. (1996). Challenges in the successful transfer of formal methods technology into industrial applications. Information and Software Technology 38(5), 313-322.
[47] Gabbar H. (2006). Modern Formal Methods and Applications. Springer.
[48] Dasso A. y Funes A. (2007). Verification, validation and testing in software engineering. Idea Group Publishing.
[49] Amman P. y Offutt J. (2008). Introduction to software testing. Cambridge University Press.
[50] Sargent R. (1999). Validation and Verification of Simulation Models. En 31st conference on Winter simulation: Simulation a bridge to the future. Phoenix, USA.

[51] Jackson D. (2001). Lightweight Formal Methods. En Formal Methods for Increasing Software Productivity Conference. Berlin, Germany.
[52] Davis J. (2005). The Affordable Application of Formal Methods to Software Engineering. ACM SIGAda Ada Letters 25(4), 57-62.
[53] Mazzola G. et al. (2006). Comprehensive Mathematics for Computer Scientists 1: Sets and Numbers, Graphs and Algebra, Logic and Machines, Linear Geometry. Springer.
[54] Agerholm S. y Larsen P. (1997). Modeling & Validating SAFER in VDM-SL. En Fourth NASA Langley Formal Methods Workshop. Hampton, USA.
[55] Lutz R. (1997). Reuse of a Formal Model for Requirements Validation. En Fourth NASA Langley Formal Methods Workshop. Hampton, USA.
[56] Janssen W. et al. (1999). Model Checking for Managers. En 5th-6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking. Toulouse, France.
[57] Goguen J. (1997). Formal methods: Promises and problems. IEEE Software 14(1), 73-85.
[58] Goguen J. (1999). Hidden algebra for software engineering. Combinatorics, Computation and Logic 21(3), 35-59.
[59] Neumann P. (1994). Computer Related Risks. Addison-Wesley.
[60] Gowen L. y Yap M. (1991). Traditional software development's effects on safety. En Sixth Annual IEEE Symposium. Amsterdam, Netherlands.
[61] Gaudel M. (1995). Testing can be formal, too. En 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development. Aarhus, Denmark.
[62] Hoare C.A.R. (2002). Assert early and assert often: Practical hints on effective asserting. Presentation at Microsoft TechFest 2002. Redmond, USA.
[63] Hierons R. et al. (2008). Formal Methods and Testing An Outcome of the FORTEST Network. Springer.
[64] Younger E. et al. (1996). Reverse engineering concurrent programs using formal modelling and analysis. En International Conference on Software Maintenance. Monterey, USA.
[65] Ward M. y Bennett K. (1995). Formal methods to aid the evolution of software. International Journal of Software Engineering and Knowledge Engineerin 5, 25-47.
[66] Hoare C.A.R. (2002). Assertions in modern software engineering practice. En 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment. Oxford, UK.
[67] Hoare C.A.R. (2003). The verifying compiler: A grand challenge for computing research. Journal of the ACM 50(1), 63-69.
[68] Hoare C.A.R. (2007). The ideal of program correctness: Third Computer Journal Lecture. Computer Journal 50(3), 254-260.
[69] George V. y Vaughn R. (2003). Application of lightweight formal methods in requirement engineering. The Journal of Defense Software Engineering 16(1), 30-38.
[70] Ghose A. (2000). Formal methods for requirements engineering. En International Symposium on Multimedia Software Engineering. Taipei, Taiwan.
[71] Meyer B. y Mandrioli D. (1992). Advances in Object-Oriented Software Engineering. Prentice-Hall.
[72] Allen R. y Garlan D. (1992). A formal approach to software architectures. En 12th World Computer Congress on Algorithms, Software, & Architecture. Madrid, Spain.
[73] Lamsweerde A. (2003). From system goals to software architecture. Lecture Notes in Computer Science 2804, 25-43.
[74] Magee J. y Kramer J. (1996). Dynamic Structure in Software Architectures. ACM Software Engineering Notes 21(6), 3–14.
[75] Allen R. (1997). A formal approach to software architecture. Doctoral Dissertation. Carnegie Mellon University
[76] Hoare C.A.R. (1975). Proof of correctness of data representations. Lecture Notes In Computer Science 46, 183-193.
[77] Dijkstra E.W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453-457.
[78] Floyd R. (1967). Assigning meanings to programs. En Symposium in Applied Mathematics. Providence, USA.

CAPÍTULO IV
Lógica y formalidad en los lenguajes formales: Un análisis en la Ciencias Computacionales

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

Originalmente la idea era buscar el papel que desempeñan los lenguajes formales en el desarrollo de la lógica, observándola como una disciplina que practica una comunidad. Pero en los primeros momentos se descubrió que el enfoque real de esta relación era muy diferente, así que la pregunta clave se orientó a encontrar la relación de la lógica y la formalidad en el desarrollo de los lenguajes formales. Se realizó un estudio descriptivo con el objetivo de encontrar la relación de la lógica y la formalidad en el desarrollo de los lenguajes formales y los resultados que se reportan en los estudios de caso. Los datos para el análisis se recogieron desde los estudios de caso publicados acerca del tema de consulta mediante un método de investigación mixta. El resultado principal es que, dadas las similitudes subyacentes con el principio de la tecnología de los Métodos Formales, la integración de la filosofía y de las perspectivas cognitivas acerca de lógica y formalidad en los lenguajes formales es un desarrollo natural. De acuerdo a la conceptualización que se presenta en esta investigación, las características más importantes de un lenguaje formal son: es escrito, su dimensión semántica no es representacional sino basada en el uso de los agentes, su función es predominantemente operativa en lugar de expresiva, y es formal porque es el producto de un proceso de de-semantificación y muestra propiedades computacionales clave.

INTRODUCCIÓN

La importancia de los lenguajes formales en las Ciencias Computacionales radica en que les permite a los desarrolladores acercarse al mejoramiento de la calidad de los productos software. Porque a pesar de que la mayoría de sus errores no genera más que algunos trastornos de menor importancia, la situación es diferente cuando se trata de sistemas críticos, tales como los utilizados en los sectores de energía nuclear, salud y aviación, en los que es esencial que sus componentes funcionen correctamente. Además, en este siglo la sociedad es *software-dependiente*, debido a que vive en una era digital masiva, de continuos desarrollos y de aparición permanente de nuevas tecnologías.

Todo esto refuerza la necesidad de que el software sea cada vez más seguro y que tenga adecuados niveles de confiabilidad. Por otro lado, la atención que los medios le prestan a los efectos de los errores de concepción en los Sistemas de Información, también juega un importante papel en la creación de expectativas en torno a la fiabilidad del software. Esto ha hecho que los gobiernos, la industria y los usuarios sean más conscientes de este problema y que, actualmente, ejerzan mayor control e incrementen las demandas por una mejor calidad en este desarrollo tecnológico. Lamentablemente, la industria del software tiene desde hace mucho tiempo la reputación de no ser capaz de lograr los compromisos de fiabilidad y de robustez que se especifican en los requisitos y, de hecho, la existencia de errores en los sistemas comerciales es tan común que la mayoría incluye la opción de reporte de error sistemático en cada versión de su producto.

Para los diseñadores de sistemas la calidad, en términos de fiabilidad y seguridad, ciertamente es una preocupación comercial. Goguen [1] enumera una serie de contratos importantes en marcha e incluso terminados por problemas de este tipo con diferentes empresas. Más allá del impacto financiero los medios recalcan la pérdida de confianza por parte de los usuarios en estas empresas, lo cual tuvo un efecto mucho más amplio y dramático que el mismo error. Pero un asunto que se debe tener en cuenta es que certificar de modo absoluto el comportamiento de un sistema es extremadamente difícil. Sin embargo, si garantizar que un sistema está totalmente libre de errores puede ser ilusorio, el grado de fiabilidad y confiabilidad que se puede tener en un producto software sigue siendo una preocupación central y bien justificada de la sociedad.

La investigación en Ciencias Computacionales y el trabajo de los científicos computacionales se ha orientado desde hace tiempo a buscarle solución a este problema, en el que un enfoque clásico para garantizar la adecuación de un sistema software es la prueba o la simulación. Esto consiste en seleccionar un conjunto de datos, considerados representativos, que se estructura y utiliza como entrada para ejecutar el sistema o un modelo funcional del mismo, para luego comparar las salidas con el conjunto de resultados esperados en la especificación. Desafortunadamente, las pruebas no pueden ser satisfactorias en su totalidad, porque para la mayoría de sistemas las combinaciones de datos de entrada son casi infinitas, lo que hace impracticable o imposible ejecutar pruebas exhaustivas.

Una noción relativa de fiabilidad, que se podría llamar *confianza*, se obtiene de una juiciosa selección de esos valores de entrada, los cuales deben impactar al mayor número posible de componentes del sistema bajo prueba. Pero la complejidad de elegirlos está directamente relacionada con la complejidad del sistema, por lo que es difícil lograrlo para sistemas complejos. Este es un punto particularmente importante cuando se discute la forma de certificar la solidez del sistema a nivel de producción, porque muchos problemas en realidad son provocados por un uso inesperado de recursos. A pesar de todo, las pruebas, como práctica, son ampliamente

aceptadas en la industria, en parte porque los organismos de certificación suelen considerar sus conclusiones como válidas. De hecho, las pruebas del software es una disciplina razonablemente bien entendida y madura cuya experiencia está integrada en este sector de la industria.

Pero existe otra alternativa, un poco menos aceptada y reconocida, que promete mejores resultados: los *Métodos Formales*. Esta tecnología constituye un enfoque matemático cuyo objetivo es garantizar la fiabilidad de los productos software, al que la industria le está dedicando cada vez mayor atención. De hecho, la necesidad de los Métodos Formales se incrementa debido a la complejidad de los sistemas actuales, a que la madurez de los mecanismos subyacentes es más adecuada para su uso industrial y a que cada año es mayor el número de investigadores y de iniciativas que surgen. Hace algún tiempo era difícil, para los no-especialistas, utilizar herramientas y Métodos Formales, pero actualmente la utilidad percibida es menos discutible [2].

En el centro de los Métodos Formales se encuentra un conjunto de formalismos y herramientas matemáticos para modelar y razonar acerca de los sistemas software y hardware, que ofrecen la posibilidad de inferir acerca del espacio de datos completo y el conjunto de configuraciones de un sistema, en lugar de en los sub-conjuntos. Por eso, en términos generales, los Métodos Formales se refieren a la utilización de los modelos matemáticos, el cálculo y la predicción, en la especificación, el diseño, el análisis y la seguridad de los sistemas computacionales. La razón por la que se les denomina *métodos* y no *modelos* matemáticos es para poner de relieve el carácter matemático involucrado [3]. Aunque la masificación de esta tecnología está condicionada por los costos, además de la experiencia necesaria para hacerlo, en las últimas décadas esta situación ha cambiado, en parte porque la inversión inicial tiende a disminuir y porque la industria está cada vez más dispuesta a apoyarlos. En este sentido la alternativa para mejorar la fiabilidad de los productos software se está moviendo lentamente de un contexto exclusivamente académico a uno industrial, particularmente en sistemas críticos [4].

Al considerar el uso de los Métodos Formales se debe tener en cuenta que no son simplemente un producto, que su utilización es mucho más que la simple instalación y aplicación de un programa cualquiera, y que se debe declarar como una cuestión estratégica y metodológica, y no solamente como un proceso técnico. Aunque la comunidad relacionada reconoce acertadamente que su uso generalizado en la industria no es una posibilidad cercana, nadie podría negar su consideración cuando la fiabilidad, la salvaguarda o la seguridad son una preocupación para el sistema. El asunto es que cada empresa de software adapta y adopta un proceso particular de desarrollo, al que es poco probable que renuncie en favor de un nuevo proceso basado en Métodos Formales, porque tendría que reformar y readaptar nuevamente su modelo de negocios.

Además, la especificación y la verificación formal no son fáciles ni baratas, aunque ese costo tiene que ser considerado a largo plazo. Por otro lado, las conclusiones tienen que tomarse con cuidado, porque esta tecnología solamente se puede utilizar para especificar o demostrar lo que de antemano estaba cuidadosamente declarado, no para razonar acerca de lo que no lo estaba, por lo que es poco probable, incluso poco razonable, hacerlo para un sistema completo. Por eso es que la práctica recomendada consiste en identificar las partes críticas del sistema y aplicar estos métodos en su diseño y validación.

En su estructura los Métodos Formales se basan en lenguajes formales, lo cual constituye el punto de partida para este capítulo con la idea de encontrar la relación entre la lógica, la formalidad y estos lenguajes en los Métodos Formales. Por supuesto, se podría decir que investigar la lógica apartada de los lenguajes formales no sería lógica propiamente dicha, pero el estudio de la historia de la lógica sugiere que esta apreciación es equivocada, porque muchas tradiciones

lógicas no usan extensivamente los lenguajes formales, aunque demuestran un nivel de sofisticación conceptual que no deja nada que desear en relación con los desarrollos actuales. Sin embargo, son muy diferentes de la investigación en lógica que se hace hoy, una cuestión que exige un análisis más dedicado, y que no es el objetivo de este trabajo.

Después de todo: ¿qué hace tan especial a los lenguajes formales y a los formalismos en general? La visión típica es que se basan en la premisa de que son, ante todo, objetos matemáticos, por lo que al tratar de definirlos con precisión surgen dificultades en precisión y complejidad técnica en las investigaciones en lógica. Pero parece que la investigación formal acerca del uso de estos lenguajes se orienta casi exclusivamente a ofrecer una explicación parcial y limitada de su impacto, lo que tergiversa de alguna manera la realidad. En el presente trabajo se adopta una concepción mucho más amplia de los lenguajes formales, con el objetivo de analizar en buena medida la relación de la lógica y la formalidad en ellos.

Originalmente la idea era buscar el papel que desempeñan los lenguajes formales en el desarrollo de la lógica, observándola como una disciplina que practica una comunidad de investigadores. Pero en los primeros momentos se descubrió que el enfoque real de esta relación era muy diferente, y que el trabajo que se difunde en la literatura se orienta mayormente a mostrar una perspectiva acerca de los lenguajes formales y sus usos en la lógica, pero vistos como artefactos cognitivos para mejorar y modificar los procesos de razonamiento de un agente. Así que la pregunta clave se orientó a encontrar la relación de la lógica y la formalidad en el desarrollo de los lenguajes formales.

Por lo tanto, los componentes clave que surgieron para el estudio fueron, en primer lugar, analizar la lógica desde una perspectiva histórica encuadrando su relación con el desarrollo de los lenguajes formales y, en segundo lugar, una investigación empírica sobre la incidencia de la formalidad en los mismos. Estos dos elementos son necesarios para encontrar una respuesta (no *la respuesta*) a la cuestión de cuál es la incidencia de la lógica y la formalidad en estos lenguajes, lo mismo que en el desarrollo y futuro de los Métodos Formales.

1. MÉTODO

En esta investigación se realizó un estudio descriptivo con el objetivo de encontrar la relación de la lógica y la formalidad en el desarrollo de los lenguajes formales y los resultados que se reportan en los estudios de caso. Un modelo descriptivo se utiliza para analizar situaciones específicas en momentos específicos, buscando comprender los estados de inicio y final a través de las modificaciones que han sufrido determinadas características.

Los datos para el análisis se recogieron desde los estudios de caso publicados acerca del tema de consulta mediante un método de investigación mixta [5]. De acuerdo con algunos autores, esta metodología les permite a los investigadores encontrar respuestas a preguntas como qué, por qué y cómo, al examinar detalladamente los resultados de casos específicos [6, 7]. Por lo tanto, en la metodología de la presente investigación se aplicaron técnicas mixtas para la recolección de datos, tales como la revisión de la literatura y el análisis de documentos, y a los resultados se aplicó un análisis mediante triangulación con el objetivo de revelar su nivel de exactitud e integración. Por su parte, los estudios de caso ayudan a comprender situaciones específicas y a analizar resultados particulares debido a que involucran una naturaleza situada y describen la complejidad de las variables involucradas. En este sentido, tienen el potencial de concretar conceptos de estudio y de contribuir a su comprensión, por lo que se suelen emplear para mostrar resultados de comparación y de evaluación de situaciones en momentos definidos.

El tema central de este trabajo son los estudios de la lógica formal reportados en los estudios de caso en la literatura, a cuyos resultados se le hizo análisis desde una perspectiva histórica encuadrando su relación con el desarrollo de los lenguajes. La búsqueda se realizó mediante un protocolo estructurado convenido por los investigadores a partir de la propuesta de Serna [8]. Posteriormente, se efectuó un análisis mediante triangulación a las variables involucradas y luego de definir la utilidad de los trabajos, se validó la información reportada mediante una verificación a la metodología y a los procedimientos aplicados. En los trabajos se verificaron los siguientes datos: 1) características del documento: año, medio divulgación, tipo, relación temática y relevancia; 2) características del autor: experiencia en el área y citaciones; 3) resultados: claridad y replicabilidad; y 4) metodología aplicada.

2. RESULTADOS

2.1 Acerca de la lógica

Frecuentemente se considera que la filosofía es una actividad de gran interés teórico, pero de poca importancia práctica. Pero la mayoría de filósofos no estaría de acuerdo con esta apreciación porque, por el contrario, muchas ideas filosóficas son necesarias para diversas actividades prácticas. La investigación de Kowalski [9-12] es un ejemplo de esta tesis, debido a que ha caracterizado el uso explícito y productivo de esas ideas. Su trabajo se ha orientado a mirar la cuestión general de hasta qué punto la filosofía ha influenciado el desarrollo de las Ciencias Computacionales. Kowalski afirma que ese influjo es muy alto y que, aunque en las primeras décadas de los computadores la relación era principalmente a través de trabajos en filosofía de las matemáticas, actualmente hay una creciente influencia de la filosofía de la ciencia, sobre todo de las relacionadas con la probabilística, la inducción y la causalidad.

Desde la perspectiva de ese período fundacional la influencia filosófica se caracterizó por la aparición y el desarrollo de tres escuelas diferentes: 1) el *logicismo*, en el que la matemática es reducible a la lógica), 2) el *formalismo*, donde se ven como el estudio de los sistemas formales, y 3) el *intuicionismo*, en el que se basan en las intuiciones de matemáticos creativos. De hecho, estas tres escuelas influenciaron el desarrollo de las Ciencias Computacionales, aunque para efectos del objetivo de este trabajo se tomará al logicismo como posición filosófica relevante. Naturalmente, este enfoque plantea un interrogante: ¿por qué los conceptos y las teorías, desarrollados por motivos filosóficos antes de los computadores, resultan tan útiles en la práctica computacional? Al parecer, aunque la lógica *inventada* por los logicistas demostró ser útil en su momento, la aplicación en las Ciencias Computacionales cambió de muchas maneras su sentido.

Por su parte, Frege [13] inició el logicismo con el objetivo de demostrar que la aritmética, no la matemática completa, era reducible a la lógica, por lo que adoptó una visión no-logicista de la interpretación kantiana de la geometría. Para lograr su objetivo ideó una manera de definir el número en términos de nociones puramente lógicas a través de un nuevo tipo de lógica formal. Posteriormente, estableció un *sistema formal* con la idea de explicar axiomas puramente lógicos y trató de demostrar que toda la aritmética se podía deducir, lógicamente, dentro de este sistema con su definición de número [14].

Pero en 1902 recibió una carta de un joven lógico llamado Bertrand Russell, en la que le demostraba que existía una contradicción en sus axiomas básicos de la lógica (lo que hoy se conoce como la paradoja de Russell). A partir de entonces, Russell trató de proporcionar nuevas ideas al logicismo y propuso la *teoría de tipos* para resolver esta paradoja. Con Alfred Whitehead desarrolló un nuevo sistema de lógica formal con el objetivo de derivar toda la matemática.

Posteriormente, Gödel [15] publicó sus *teoremas de la incompletitud* en los que demostraba que no todas las verdades aritméticas se pueden derivar con la teoría de Russell y Whitehead [16] y que, por lo tanto, no cumple con su objetivo de reducirla a la lógica. Además, demostró que los resultados de su trabajo no se aplicaban solamente a esta teoría, sino a cualquier sistema logicista similar.

Por otro lado, Hilbert [17] desarrolló la filosofía formalista de las matemáticas, se hizo cargo del concepto de *sistema formal* de los logicistas y, contrariamente a ellos, sugirió que se podría construir un sistema formal axiomático diferente para cada rama de las matemáticas. El trabajo de Frege había demostrado que existía el peligro de una contradicción en un sistema formal, por lo que, para evitarla, Hilbert sugirió que la consistencia de los sistemas formales se debe probar utilizando únicamente métodos informales simples de aritmética finita, pero el segundo teorema de Gödel demostraba que tales pruebas no se daban para todas las ramas importantes de la matemática. De esta manera Hilbert había demostrado que dos de las tres posiciones principales en la filosofía de las matemáticas eran insostenibles.

En cuanto a la tercera escuela, el *intuicionismo*, cuyos principios no habían sido refutados por los teoremas de la incompletitud de Gödel, tenía otras dificultades que la hacían inaceptable para la mayoría de los matemáticos. Un trabajo sistemático, por fuera de la idea de que la matemática era la construcción intuitiva de matemáticos creativos, parecía indicar que algunas de las leyes lógicas asumidas en las matemáticas, especialmente la ley del tercero excluido, no tenían una debida justificación. Por lo tanto, los intuicionistas crearon un nuevo tipo de matemáticas, que no implicara a ésta ni a otras leyes sospechosas, pero resultó ser más complicada y compleja que la tradicional, por lo que fue rechazada por la mayoría.

Hasta entonces se aceptaba que los teoremas de Gödel daban inició a un período de depresión en la filosofía de las matemáticas, porque las tres principales escuelas parecían haber fracasado. Sin embargo, en el período posterior a la guerra de ideas se descubrió que eran muy útiles para el desarrollo y la expansión de las Ciencias Computacionales. El cálculo de predicados de Frege se convirtió en una de las herramientas teóricas más utilizadas en estas ciencias, especialmente en la demostración automática de teoremas. Robinson [18] desarrolló una forma de cálculo de predicados diseñado específicamente para demostrar teoremas mediante computador, que también resultó ser útil en otras aplicaciones de la lógica en la computación. Robinson explica que la lógica diseñada para el uso por computador puede ser diferente de otra para el uso humano, y afirma que una regla de inferencia que requiere gran cantidad de cómputo no plantea ningún problema para una máquina, pero sí para una persona.

Por otro lado, para aplicaciones computacionales es deseable reducir tanto como sea posible el número de reglas de inferencia. Si un sistema tiene un alto número de reglas simples, un ser humano, dotado con cierta habilidad intuitiva, podría ver cuáles serían más apropiadas para una situación particular. Mientras que un computador, a falta de esa habilidad, puede que tenga que probar cada una antes de encontrar la más apropiada. Así que una lógica para los seres humanos puede tener gran número de reglas de inferencia simples, mientras que una para computador sería mejor con menos reglas, pero más complicadas. El punto importante a señalar aquí es que en cuanto a los sistemas lógico-lingüísticos los requisitos de un computador pueden ser muy diferentes de los de un ser humano.

El trabajo de Kowalski [9] y el de Colmerauer para el desarrollo del lenguaje de programación PROLOG se basaron en Robinson [19]. Además, el concepto de la programación lógica inductiva de Muggleton [20] se originó en la idea de invertir la lógica deductiva de Robinson para producir

una lógica inductiva. El predecesor de Frege, Boole, había introducido también un sistema de lógica formal, y Jevons, con su propia versión de la lógica de Boole e influenciado por Babbage, había construido una máquina para realizar inferencias lógicas. Por su parte, Church [21] había previsto su primera versión del λ-cálculo y proporcionó una nueva base para la lógica en el estilo de Russell y Whitehead, aunque, como lo demostraron Kleene y Rosser [22] usando una variación de la paradoja de Richard, resultaron ser incompatibles. A pesar de este contratiempo el λ-cálculo se modificó y resultó ser muy útil en las Ciencias Computacionales, al convertirse en la base de lenguajes de programación tales como LISP, Miranda y ML, y de hecho se utiliza como una herramienta básica para el análisis de otros lenguajes de programación. Por supuesto la teoría de tipos utilizada en la computación contemporánea no es lo mismo que la teoría original de Russell, no obstante, son descendientes de su sistema original.

Las ideas de Robinson [18] acerca de las diferencias entre los requisitos de los seres humanos y de los computadores en relación con los sistemas lógico-lingüísticos ayudaron a explicar esta situación. Un sistema con variables de diferentes tipos es incómodo e inconveniente para las personas y realmente no confiere ninguna ventaja. En la mayoría de casos puede que no cometan errores en las fórmulas ya que su comprensión intuitiva les impediría escribir tonterías. Pero con los computadores pasa lo contrario debido a no tienen ningún problema para manejar diferentes variables de muchos tipos, además, sin la guía proporcionada por una sintaxis de tipo estricto fácilmente pueden producir fórmulas sin sentido, debido a que carecen de cualquier comprensión intuitiva del significado deseado de la misma. Esto demuestra que la lógica en las Ciencias Computacionales es una herramienta importante para la verificación de software y hardware; además, en el desarrollo de software su influencia no se limita a los lenguajes de programación específicamente lógicos, porque de hecho proporciona el núcleo sintáctico de los lenguajes más comunes [23], entre los que Begriffsschrift [24] es el primero completamente formalizado y, en cierto sentido, el precursor de todos ellos [25].

Ahora bien, si la investigación de Frege y Russell fue motivada por consideraciones filosóficas y fueron influenciados solamente de manera desdeñable por la computación, entonces ¿por qué su trabajo en lógica ha resultado tan útil en las Ciencias Computacionales? Porque antes de ellos las matemáticas eran descritas como *semiformales*, aunque, por supuesto, se utilizaba el simbolismo incorporado en el lenguaje natural. El uso de ese lenguaje informal generaba ambigüedades en los conceptos, lo que creaba errores y confusiones, pero Frege, al buscar la certeza, pensó que podía mejorar las cosas mediante la formalización, con la que, al definir la matemática con precisión, se evitaría estos problemas. Además, las demostraciones se tendrían que estructurar en pasos desglosados, de manera que cada uno aplicara una regla lógica simple y, obviamente, correcta, de tal manera que se pudiera eliminar cualquier asunto intuitivo [26].

Los métodos que Frege utilizó en su búsqueda de la certeza en matemáticas crearon un sistema adecuado para las Ciencias Computacionales, porque lo que hacen es mecanizar el proceso para comprobar la validez de una prueba. Pero si esa prueba está escrita con características humanas semi-formales, entonces su validez no puede ser verificada mecánicamente, porque se necesita un matemático que aplique su intuición para revisar cada línea. Sin embargo, cuando esa prueba se formaliza su comprobación se convierte en una cuestión puramente mecánica de utilizar un conjunto de reglas de inferencia prescrito.

Aunque Frege y sus sucesores logicistas estaban buscando demostraciones matemáticas que una máquina llevará a cabo mediante la descomposición en pasos sencillos, desde un punto de vista filosófico Frege y Russell se involucraron en el proyecto de la mecanización del pensamiento y, debido a que vivían en una sociedad mecanizada con una cantidad cada vez mayor de trabajo

mental, el desarrollo de su proyecto fue casi natural. Además, era igualmente natural elegir a las matemáticas para iniciar el proceso de mecanización, porque dada su historia lógica, y a diferencia de otras áreas del conocimiento, se suponían parcialmente formalizadas. Estas consideraciones quizá explican la importancia de la filosofía de las matemáticas en estos días, pero, así como los pensadores se ven presionados a seguir adelante con la mecanización de las matemáticas, también ha habido quienes se oponen y se orientan más por los aspectos humano-intuitivos de las mismas. Aunque esta línea de pensamiento es en muchos aspectos reaccionaria y no ha detenido los avances de la mecanización, hay algo de verdad en ella, porque siempre que las matemáticas sigan siendo hechas por humanos será evidente la necesidad de retener parte de sus características intuitivas. Esta es otra razón por la que los logicistas, aunque pensaban que estaban construyendo una base segura para las matemáticas, de hecho, creaban una adecuación de las matemáticas para las Ciencias Computacionales.

2.2 Acerca de los lenguajes

El estudio de los lenguajes se refiere a la investigación razonada acerca de los orígenes, la naturaleza del significado, el uso, la cognición y la relación entre ellos y la realidad. Esto coincide en cierta medida con el estudio de la epistemología, la lógica, la filosofía de la mente y campos tales como la lingüística y la psicología, aunque para muchos filósofos analíticos es una disciplina en su propio derecho que, como área de investigación, realiza preguntas como: ¿qué es el significado? ¿En qué lenguaje se representa el mundo real? ¿El lenguaje se aprende o es innato? ¿De qué manera las partes le dan significado a una frase?

En la historia es posible encontrar referencias acerca del estudio del lenguaje que se remontan a 1500 de nuestra era, mucho antes que se hiciera cualquier descripción sistemática de los mismos. En la filosofía medieval temprana hubo varias escuelas de pensamiento donde se discutía cuestiones lingüísticas. En la tradición occidental los primeros trabajos en este tema fueron cubiertos por Platón, Aristóteles y los Estoicos de la antigua Grecia, en los que, por ejemplo, Platón consideraba que los nombres de las cosas estaban determinados por la naturaleza y que cada fonema (unidad estructural más pequeña que distingue significado) representaba las ideas básicas o sentimientos.

Aristóteles sostenía que el significado de un predicado (la forma en que el sujeto se modifica o describe en una oración) se establece a partir de una abstracción de las similitudes entre varias cosas individuales (nominalismo); además, presumía que esas similitudes estaban constituidas por una forma de comunalidad real, sin embargo, también fue un defensor del realismo moderado. Los Estoicos hicieron importantes contribuciones al análisis de la gramática y distinguían cinco partes en un discurso: 1) nombres, 2) verbos, 3) calificativos, 4) conjunciones, y 5) artículos; además, lo que llamaban la *Lekton* (significado o sentido de cada término) dio lugar al concepto de proposición de una frase, es decir, la capacidad para ser considerada como una afirmación que puede ser verdadera o falsa.

Los Escolásticos de la Edad Media, provocados en cierta medida por la necesidad de traducir textos griegos al latín, se interesaron en las sutilezas del lenguaje y su uso. Consideraban que la lógica debía ser una ciencia del lenguaje y anticiparon muchos de los problemas de la investigación moderna del mismo, incluyendo los fenómenos de vaguedad y ambigüedad, las doctrinas de la suposición correcta e incorrecta (interpretación de un término en un contexto específico), lo mismo que el estudio de palabras y términos categoremáticos (con significado propio) y sincategoremáticos (sin significado propio). Los lingüistas del Renacimiento, impulsados por el descubrimiento gradual en occidente de los caracteres chinos y los jeroglíficos egipcios,

estaban particularmente interesados en la idea de un lenguaje filosófico o lenguaje universal. Después de todo esto el lenguaje comenzó a desempeñar un papel más central en la filosofía occidental del siglo XIX y, más aún, en el XX, especialmente después de la publicación del trabajo de Ferdinand de Saussure [27]. Una de las cuestiones fundamentales planteadas en esa investigación se refiere a la búsqueda del significado general del término.

Por otro lado, y de acuerdo con la semiótica (estudio de los signos procesados en la comunicación y de cómo se construye y se entiende el significado), el lenguaje es la manipulación y el uso de símbolos con el fin de llamar la atención sobre el contenido-significado, en cuyo caso los seres humanos no serían los únicos poseedores de habilidades lingüísticas. La lingüística es el campo de estudio que hace preguntas tales como: ¿en qué se diferencia un lenguaje particular de otro? ¿Qué es lo que hace al español un lenguaje? ¿Cuál es la diferencia entre el inglés y el francés? Además, lingüistas como Chomsky [28] hacen hincapié en el papel de la gramática y la sintaxis (reglas que gobiernan la estructura de las oraciones) como características de cualquier lenguaje, por lo que afirma que los humanos nacen con una comprensión innata de lo que llama gramática universal (principios lingüísticos compartidos por todos los seres humanos) y que la exposición de un niño a un lenguaje en particular solamente desencadena este conocimiento antecedente.

La traducción y la interpretación presentan otro problema al estudio del lenguaje, por lo que, a mediados del siglo XX, Van Orman [29] argumentó a favor de la indeterminación del significado y la referencia con base en el principio de la traducción radical, por ejemplo, cuando se intenta traducir desde un lenguaje indeterminado. Afirmaba que en tal situación es imposible, en principio, estar absolutamente seguro del significado o la referencia en un enunciado que trata un hablante en un lenguaje, como el de una tribu primitiva, y puesto que las referencias son indeterminadas habría muchas interpretaciones posibles, pero ninguna más correcta que las otras. La visión resultante se llama holismo semántico, que sostiene que el significado no es algo que se asocie con una sola palabra o frase, porque solamente se puede atribuir a un lenguaje completo. Davidson [30] amplió este argumento a la noción de interpretación radical, en el sentido de que el significado que cualquier individuo le atribuye a una frase puede determinarse únicamente mediante la atribución de significados a muchas, quizás todas, las afirmaciones individuales, así como de sus estados y actitudes mentales.

En este sentido es posible afirmar que no es fácil definir qué es significado, pero se acepta que puede ser el contenido que transportan las palabras o signos intercambiados por las personas, cuando se comunican a través de un lenguaje. Lingüísticamente podría decirse que existen dos tipos de significado esencialmente diferentes: 1) el *conceptual*, que se refiere a las definiciones de las palabras mismas y cuyas características pueden ser tratadas utilizando el análisis de rasgos semánticos, y 2) el *asociativo*, que se refiere a las comprensiones mentales individuales del orador y que pueden ser connotativo, colocativo, social, afectivo, reflejado o temático. Por otro lado, existen varios enfoques diferentes para explicar el significado lingüístico:

- *Teorías*, que afirman que los significados son contenidos puramente mentales provocados por los signos. Este enfoque se asocia principalmente con la tradición empirista británica de John Locke [31], George Berkeley [32] y David Hume [33], a pesar de que el interés en ella ha sido renovado por algunos teóricos contemporáneos bajo el disfraz del *internalismo semántico*.

- *La verdad condicional*, en la que el significado son las condiciones en que una expresión puede ser verdadera o falsa. Esta tradición se remonta a Gottlob Frege [13], aunque también ha habido amplio trabajo moderno en esta área.

- *El uso*, en el que se entiende que el significado involucra o está relacionado con los actos del habla y las expresiones particulares, no con las expresiones como tal. Este enfoque fue iniciado por Ludwig Wittgenstein [34] en su visión comunitaria del lenguaje.

- *El externalismo semántico*, en el que se entiende el significado como un equivalente a las cosas en el mundo que están conectadas a los signos. Tyler Burge [35] y Saul Kripke [36] son los defensores más conocidos de este enfoque.

- *El verificacionismo*, que asocia el significado de una frase con su método de verificación o falsificación. Este enfoque fue adoptado por los positivistas lógicos de principios del siglo XX.

- *El pragmatismo*, que sostiene que el significado o la comprensión de una oración está determinado por las consecuencias de su aplicación. Este enfoque fue acogido por Charles Peirce [37] y otros pragmáticos de principios del siglo XX.

Otro concepto importante en el estudio del lenguaje es la *intencionalidad*, definida a veces como la temática. Hay cosas que tratan acerca de otras cosas, por ejemplo, una creencia puede ser acerca de los milagros, pero un milagro no trata acerca de nada; un libro o una película pueden tratar sobre New York, pero el mismo New York no tiene nada que ver. La intencionalidad es el término para esa función en la que ciertos estados mentales tienden a tratar sobre objetos y estados de cosas en el mundo real. Por lo tanto, las creencias, miedos, esperanzas y deseos son intencionales en el sentido de que un objeto los debe tener. El término intencionalidad fue acuñado inicialmente por los escolásticos de la Edad Media, pero revivido por Brentano [38], un precursor de la escuela fenomenológica que afirmaba que todos fenómenos, y no solamente los mentales, exhiben intencionalidad, para lo que utilizó como prueba que los fenómenos mentales no podían ser lo mismo que los fenómenos físicos.

Todo esto hizo que los estudiosos posteriores se plantearan preguntas: ¿cómo la mente y el lenguaje imponen intencionalidad en los objetos que no son intrínsecamente intencionales? ¿Cómo se representa los estados mentales y cómo hacen para representar objetos del mundo real? Cuyas respuestas se establecieron en la teoría de los actos ilocutorios y en la teoría del discurso, en las que se observa al lenguaje como una forma de acción y de comportamiento humano, por lo que al *decir* algo lo que realmente se logra es *hacer* algo. Combinando esta idea con la intencionalidad, John Searle [39] concluye que las propias acciones tienen un tipo de intencionalidad.

Debido a que el lenguaje interactúa con el mundo, a lo que los investigadores llaman referencia, durante años ha interesado a muchos filósofos y lingüistas. Entre ellos, John Mill [40] creía en un tipo de referencia directa en la que el significado de una expresión reside en lo que se señala en el mundo. Identificó dos componentes significativos para la mayoría de términos de un lenguaje: 1) la *denotación* o significado literal de una palabra o término, y 2) la *connotación* o coloración cultural y/o emocional subjetiva unida a una palabra o término. Para él, los nombres propios tienen solamente una denotación y no tienen connotación, mientras que una sentencia que hace referencia, por ejemplo, a una criatura mítica, no tiene sentido y no es ni verdadera ni falsa, porque no tiene ningún referente en el mundo real.

Por su parte, Frege [13] fue un defensor de la teoría de la referencia mediada, que postula que las palabras se refieren a algo en el mundo externo, pero insistía en que hay más en el significado de un nombre que simplemente el objeto a que se refiere. Frege divide el contenido semántico de cada expresión, incluyendo las frases, en dos componentes: 1) *sinn*, generalmente traducido como

sentido, y 2) *bedeutung,* que es significado, denotación o referencia. El sentido de una oración es el pensamiento abstracto, universal y objetivo que expresa, además del modo de presentación del objeto al que se refiere. La referencia es el objeto u objetos que escogen las palabras en el mundo real y que representa un valor de verdad (verdadero o falso), a la vez que determina el sentido y los nombres que se refieren al mismo objeto, aunque pueden tener diferentes sentidos.

Bertrand Russell también fue un descriptivista, y sostenía que los significados o contenidos semánticos de los nombres eran idénticos a las descripciones que les asocian los oradores, y que una descripción contextual apropiada puede ser sustituida por un nombre. Pero afirmaba que solamente las expresiones referenciadas directamente eran nombres lógicamente propios, tales como *yo*, *ahora*, *aquí* y otros deícticos (términos que simbólicamente señalan o indican un estado de las cosas). Refería los nombres propios como descripciones abreviadas definidas o nombres base para una descripción más detallada, y consideraba que como tal no era significativa por su cuenta, por lo que no era directamente referencial.

Saul Kripke [36] argumenta en contra del descriptivismo con el argumento de que los nombres son designadores rígidos, que se refieren a la misma cosa en cada mundo posible en el que existan. En este sentido, y para explicar la relación entre las partes significativas y el conjunto de frases, la semántica filosófica (estudio del significado en el lenguaje) tiende a centrarse en el principio de la composicionalidad. Este principio afirma que una frase se puede entender sobre la base del sentido de sus partes (palabras o morfemas), junto con una comprensión de su estructura (sintaxis o lógica). Por lo tanto, el significado de una expresión compleja está determinado por los significados de sus expresiones constituyentes y las normas utilizadas para combinarlas.

Las funciones también se pueden utilizar para describir el significado de una frase, debido a que una función proposicional es una operación del lenguaje que toma una entidad (sujeto) como entrada y un hecho semántico (proposición) como salida. En este sentido, existen dos métodos generales para comprender la relación entre las partes de una cadena lingüística y la forma en que se juntan: 1) los *árboles sintácticos*, que se centran en las palabras de una frase teniendo en mente la gramática de la misma, y 2) los *árboles semánticos*, que se centran en el rol del significado de las palabras y en cómo se combinan. Además, tres principales escuelas de pensamiento sobre la cuestión del aprendizaje del lenguaje: 1) el *conductismo*, que sostiene que la mayor parte del lenguaje se aprende a través de condicionamiento, 2) la *prueba de hipótesis*, según la cual el aprendizaje ocurre a través de la postulación y la prueba de hipótesis, y usando la facultad general de inteligencia, y 3) el *innatismo*, que sostiene que por lo menos alguna de las configuraciones sintácticas es innata y cableada con base en ciertos módulos del cerebro.

También, y cuando se trata del lenguaje, se ha propuesto diversos modelos acerca de la estructura del cerebro:

- Los *conexionistas*: hacen hincapié en la idea de que el léxico de una persona y sus pensamientos operan en una especie de red asociativa distribuida.
- Los *nativistas*: aseguran que hay dispositivos especializados en el cerebro que se dedican a la adquisición del lenguaje.
- Los *computacionales*: enfatizan en la noción de un lenguaje representacional del pensamiento y de la lógica como un procesamiento computacional que la mente realiza sobre ellos.
- Los *emergentistas*: se centran en la idea de que las facultades naturales son un sistema complejo que emerge de partes biológicas más simples.

- Los *reduccionistas*: intentan explicar los procesos mentales de alto nivel en términos de la actividad neurofisiológica de bajo nivel del cerebro.

También hay tres argumentos principales con respecto a la relación entre el lenguaje y el pensamiento: 1) el de Edward Sapir [41], Benjamin Whorf [42] y Michael Dummett [43], entre otros, que sostiene que el lenguaje es analíticamente anterior al pensamiento; 2) el de Paul Grice [44] y Jerry Fodor [45], para el que el pensamiento y el contenido mental tienen prioridad sobre el lenguaje, por lo que el lenguaje hablado y el escrito derivan su intencionalidad y significado de un lenguaje interno codificado en la mente, sobre todo teniendo en cuenta que la estructura de los pensamientos y la estructura del lenguaje parecen compartir un carácter sistemático composicional; y 3) el que sostiene que no hay manera de explicar una cosa sin la otra.

Por otro lado, la mayoría de investigadores y filósofos han sido más o menos escépticos acerca de la formalización de los lenguajes naturales y, por lo tanto, del uso de la lógica formal para analizarlos y entenderlos. Aunque algunos, como Alfred Tarski [46], Rudolf Carnap [47], Richard Montague [48] y Donald Davidson [49], han desarrollado lenguajes formales o han formalizado partes del lenguaje natural en su investigación. Incluso otros, como Paul Grice [50], han negado que exista un conflicto sustancial entre la lógica y el lenguaje natural.

Sin embargo, Gilbert Ryle [51] y Peter Strawson [52] hacen hincapié en la importancia de estudiar el lenguaje natural sin tener en cuenta las condiciones de verdad de las frases y las referencias de los términos. Para ellos el lenguaje es algo totalmente diferente a la lógica, y cualquier intento de formalizarlo utilizando las herramientas de la lógica está condenado al fracaso. Ryle desarrolló la teoría de los actos del orador, en la que describe el tipo de cosas que se puede hacer con una frase (afirmación, mandato, investigación y exclamación), en diferentes contextos de uso y en diferentes ocasiones; mientras que Strawson argumenta que la semántica de las tablas de verdad de los conectores lógicos no captura el significado de sus contrapartes en el lenguaje natural.

Algunos filósofos niegan que los computadores puedan entender un lenguaje, pero de acuerdo con la teoría del significado de Wittgenstein [34] parece que sí pueden hacerlo. Para demostrarlo basta con tomar su ejemplo y sustituir al trabajador por un computador: como a la máquina es posible darle órdenes mediante, por ejemplo, un programa, se espera que pueda llevar a cabo las instrucciones y, por lo tanto, es sensato afirmar, al igual que en el caso humano, que las comprende. En ese caso el intercambio se da a través de un lenguaje y, de acuerdo con Wittgenstein, se está utilizando correctamente los símbolos involucrados, por lo que se entiende su significado. De manera similar se puede afirmar que los animales entienden algunas palabras del lenguaje natural, porque al darle órdenes llevan a cabo la acción esperada, es decir, entienden los símbolos.

Pero en lo que se refiere al lenguaje hay una diferencia significativa entre los animales y los computadores, porque los primeros solamente pueden entender comandos que consisten esencialmente de un símbolo, y la gramática va mucho más allá de eso. Los computadores por su parte son mucho más meticulosos que los humanos acerca de la gramática, porque *hablan* sin gramática y, no obstante, sus expresiones se pueden entender, mientras que no comprenderá las instrucciones que contienen algún error sintáctico simple.

Esto obligatoriamente revive el tema central de las diferentes necesidades lingüísticas de los computadores y las personas, porque para los primeros es más fácil entender los lenguajes formales precisos, que para los seres humanos son difíciles. Por el contrario, los humanos encuentran más fácil entender los lenguajes naturales informales, que son completamente

oscuros para los computadores. Aunque estas máquinas pueden realizar cosas maravillosas, hay que darles las órdenes en un lenguaje que puedan entender, lo que genera un círculo vicioso en las Ciencias Computacionales debido a que el lenguaje, que para los humanos es fácil, para ellos es difícil, mientras que el que para ellos es fácil para los humanos es difícil.

Entonces, se necesita un lenguaje que se pueda utilizar desde ambos lados, un dilema en el que la lógica formal demuestra su utilidad. Este lenguaje es intermedio entre el código máquina y el lenguaje natural, porque tiene la sintaxis precisa que hace que sus oraciones sean comprendidas por los computadores, a la vez que tiene suficiente semejanza con el lenguaje natural para ser comprensible por los seres humanos, aunque con entrenamiento. Incluso, y como señala Robinson [18], dentro de la misma lógica hay algunas formulaciones que son más adecuadas para las máquinas y otras que lo son para los seres humanos, pero en términos generales la lógica formal es un lenguaje intermedio de los más adecuados para ambos mundos, por lo que facilita la interacción humano-computador y es la razón principal por la que ha demostrado conveniencia en las Ciencias Computacionales.

2.3 Acerca de los lenguajes formales

El desarrollo histórico de los lenguajes formales se puede ver como un proceso de evolución cultural, en el que los humanos desarrollan herramientas para realizar ciertas tareas y para resolver ciertos problemas de manera eficiente [53]. Naturalmente, el proceso de búsqueda solamente puede tener lugar en el ámbito de las limitaciones inherentes al aparato cognitivo humano, pero en estos desarrollos, y al igual que en los procesos de la evolución biológica, el azar y la contingencia influyen de alguna manera. Al mismo tiempo, esto no supone una forma de relativismo científico-cultural, porque de hecho el desarrollo de estos lenguajes constituye un progreso real [54]. Un formalismo es una poderosa tecnología que le permite a los seres humanos razonar en formas que, de otro modo, estarían prácticamente fuera de su alcance. La idea subyacente es que su desarrollo histórico se entiende mejor desde el punto de vista del marco de la cognición extendida y, de hecho, parece ser que la historia de las notaciones matemáticas se inscribe en el punto de vista del concepto, pero este ambicioso objetivo está por fuera de la presente investigación.

El desarrollo de estos lenguajes tiene una relación directa con el de las técnicas algorítmicas y algebraicas en el cálculo del mundo árabe, que fueron traídas a Europa por las escuelas del ábaco. Sin este vínculo no es posible entender el progreso de la notación matemática de los siglos XVI y XVII, iniciadas por Viète y validadas por Descartes. Sin embargo, para entender los acontecimientos relevantes también hay que tener en cuenta la evolución de la lógica en India, China y el mundo árabe, por lo que, tal como lo describe Staal [53], la historia de los lenguajes formales se extiende a través de miles de años y, por lo menos, en dos continentes. Además, es necesario examinarlos en el contexto del desarrollo de las notaciones matemáticas, particularmente en la búsqueda de aquellas que podrían servir como instrumentos de cálculo [55]. De hecho, muchos de los acontecimientos centrales se llevaron a cabo dentro de las matemáticas, y solamente hasta el siglo XVII fueron importados desde la lógica académica. Es decir, desde su nacimiento en la antigua Grecia hasta el siglo XVII la lógica académica fue básicamente práctica, con un cuerpo estable de dispositivos y técnicas *notacionales*, particularmente letras esquemáticas y regimentación.

Con el tiempo se ha aceptado que los lenguajes formales son un cuerpo de eslabones y símbolos de un alfabeto finito, y que las gramáticas formales sirven como dispositivos para definirlos, por lo que, dependiendo del contexto, estos eslabones y símbolos constituyen un lenguaje en el que

pueden ser referidos como palabras, frases, programas, etc. Se acepta entonces que todos los lenguajes escritos, naturales, de programación, o de cualquier otro tipo, están orientados por esta idea. Por otro lado, aunque los lenguajes formales pueden ser entendidos bajo esta forma general, hay que imponerles mayor estructura e ir más allá de una imagen lineal de eslabones.

Entonces: ¿cómo especificar un lenguaje formal? En principio, y si se tratara de un conjunto finito de eslabones, se podría lograr haciendo una lista de sus elementos. Pero con los lenguajes infinitos la situación es diferente, porque hay que inventar un dispositivo *finitista* para producirlos. Tales dispositivos pueden ser las gramáticas, los sistemas de re-escritura, los autómatas, entre otros. De hecho, una parte de la teoría de los lenguajes formales se puede ver como el estudio de dispositivos *finitistas* para generar lenguajes infinitos. En todo caso, lo que hoy se conoce como *teoría de lenguajes formales* tuvo diferentes orígenes:

1. Las *matemáticas*, particularmente algunos problemas en el álgebra combinatoria y en la de semi-grupos y monoides. A principios del Siglo XX Axel Thue [56, 57] investigó los patrones evitables e inevitables en las palabras largas e infinitas, e introdujo la noción formal de un sistema de re-escritura o gramática, que posteriormente refrendó Emil Post [58].
2. La *lógica*, que de acuerdo con la terminología actual sería la teoría computacional y que tiene como principal referente al trabajo de Alan Turing [59], cuya idea general era encontrar modelos computacionales. Desde esta perspectiva el poder de un modelo específico se puede describir por la complejidad de los lenguajes que genera o acepta.
3. Es natural que gran parte de la teoría de los lenguajes formales tenga origen en la *lingüística* y, de hecho, se puede rastrear en el tiempo; uno de los estudios más representativos a este respecto es el de las gramáticas de Noam Chomsky [60].
4. Otra parte de la teoría tiene origen en el *modelado* de ciertos objetos o fenómenos. Un modelo se puede expresar por, o identificar con, un lenguaje, y sus tareas específicas dan lugar a tipos específicos de lenguajes [61]. Esto significa que el lenguaje utilizado no consiste de palabras, sino de árboles, grafos u otros *objetos multi-dimensionales*.

Actualmente, el estudio de los lenguajes formales constituye una importante sub-área de las Ciencias Computacionales, que surgió en los años 50 cuando Noam Chomsky [60] presentó un modelo matemático de una gramática en relación con su estudio de los lenguajes naturales. Poco después, cuando se definió la sintaxis del lenguaje ALGOL mediante una gramática libre de contexto, se encontró que ese concepto era importante para la programación. Un desarrollo que condujo naturalmente a la compilación dirigida por sintaxis y al concepto de un compilador de compiladores; desde entonces la investigación ha trabajado en armonía con la teoría de autómatas, al punto de que hoy es casi imposible tratarlas por separado. Además, generalmente se acepta que ningún estudio serio en Ciencias Computacionales estaría completo sin un conocimiento de las técnicas y los resultados de ambos.

En todo caso los lenguajes formales se originan en los formalismos de notación simbólica de las matemáticas, especialmente en la combinatoria y la lógica simbólica, a las que, posteriormente, se adicionaron diversos códigos necesarios para el cifrado de datos, la transmisión y la corrección de errores y, particularmente, diversos modelos matemáticos de la automatización y la computación. Sin embargo, fue solamente después de que se conocieran las ideas innovadoras de Chomsky y el enfoque de algebra combinatoria cuando la teoría de los lenguajes formales tuvo realmente su apogeo, que luego se reforzó con la fuerte influencia de los lenguajes de programación. Aunque en los días de gloria de los lenguajes formales, entre 1960 y 1970, gran

parte de la investigación estuvo orientada a la teoría, hoy se puede decir que se ha equilibrado con la práctica y es un campo de estudio con amplia actividad.

En este sentido, las Ciencias Computacionales teóricas estudian el fundamento matemático de la computación e investigan el poder y las limitaciones de los dispositivos informáticos. Además, para utilizar pruebas matemáticas rigurosas se necesita modelos matemáticos abstractos, lo más simples posible, para analizarlos con facilidad, pero lo suficientemente potentes como para realizar procesos de cálculo relevantes. La razón detrás de su éxito es el hecho de que los computadores se pueden modelar con máquinas abstractas simples, en las que estos modelos ignoran los detalles de implementación y se concentran en procesos computacionales reales.

Por otro lado, todas las máquinas abstractas manipulan cadenas de símbolos, por lo que el poder de cualquier modelo computacional se puede determinar mediante el análisis de la complejidad de los lenguajes formales que puede describir. Por ejemplo, los autómatas finitos solamente pueden definir lenguajes muy simples, llamados lenguajes regulares, mientras que los autómatas de pila describen lenguajes libres de contexto más complicados.

Entonces: ¿cuál es la razón de ser de la elaboración y adopción de los lenguajes formales? ¿Cuál será la rentabilidad de la inversión en el aprendizaje de esta técnica? Hoy se tiene la tendencia a tomar por sentado el uso de la lógica en los lenguajes formales y a no inquietarse de por qué se hace. Sin embargo, en el desarrollo de los lenguajes pioneros hubo que invertir tiempo para justificar su uso, fundamentalmente con base en tres de sus principales funciones:

1. La *expresividad*. Quizás la justificación más frecuente y más profundamente arraigada para la adopción de estos lenguajes es la insuficiencia de los tradicionales en cuanto a expresividad y comunicación, particularmente en lo que se refiere al lenguaje natural. Esta insuficiencia suele atribuirse a sus imperfecciones, tales como la falta de precisión y claridad, la presencia de ambigüedades, y otras tantas. Desde este punto de vista se mantiene posiciones como que los lenguajes formales simplemente extienden, pero no reemplazan, a los comunes; que representan una mejora en cuanto a que permiten realizar ciertas tareas con mayor facilidad, pero que podrían, al menos en teoría, realizarse sin ellos; o que tienen el propósito de reformar a los comunes [62].

   En el pasado reciente la idea de que los lenguajes formales remedian de alguna manera las deficiencias en los comunes tiene sus orígenes en el Begriffsschrift de Frege, cuyo objetivo fue proporcionar fundamentos lógico-matemáticos. Para eso tuvo que explicar las pruebas matemáticas de una forma completamente lúcida, destacando todas las medidas inferenciales y todas las suposiciones hechas, por lo que su logicismo motivó la formulación de un lenguaje formal puramente lógico que se aísla de la no-lógica (intuitiva) para las pruebas. Dada la estructura sistemática y lógicamente imperfecta del lenguaje natural, la declaración de sus reglas formales de formación y transformación de palabras es tan complicada que, en la práctica, difícilmente sería factible. Pero la misma dificultad se puede plantear en los lenguajes formales que, a pesar de evitar ciertas imperfecciones lógicas que caracterizan al natural, deben, necesariamente, ser todavía más complicados desde el punto de vista lógico, debido al hecho de que son lenguajes conversacionales y, por lo tanto, todavía dependen del natural [63].

2. La *representación*. La preocupación general por la expresividad también ha llevado a los teóricos a formular la idea de que los lenguajes formales se deben esforzar por ser representaciones icónicas de lo que sea que representen. Estas consideraciones se pueden encontrar en Leibniz [64], quien sugirió que los símbolos individuales como característica

deben ser claramente icónicos. Naturalmente, la idea de que los símbolos en un lenguaje formal deban ser como jeroglíficos es ingenua, porque su finalidad es diferente a la del lenguaje natural, en el sentido de que tienen su mayor aplicación en la prueba, lo que los habilita para ordenar el trabajo de los autómatas. Por lo tanto, incluso si existiese retos filosóficos que sustenten la concepción de los lenguajes formales como dispositivos icónicos, parece que hay evidencia empírica que corrobora la idea de que los agentes humanos, de hecho, tienen una relación esquemática icónico-cognitiva con los formalismos.

3. El *cálculo*. Esta función a menudo se atribuye a los lenguajes formales, particularmente por el papel que desempeñan los signos físicos en las operaciones relacionadas. De hecho, para Leibniz todas las formas de pensamiento son variaciones de los procesos de cálculo, y todos los cálculos implican la manipulación de símbolos. Entonces, un sistema de notación bien diseñado debe ser, básicamente, una herramienta ideal para el razonamiento, lo que permitiría el establecimiento de las verdades, incluso más allá del dominio de las matemáticas.

   Si esto se hace, es decir, si se formula una característica universal, siempre que se presente controversias no habría necesidad de las discusiones entre filósofos, sino entre matemáticos. Para ello sería suficiente con tomar un lápiz y sentarse con el ábaco a realizar la tarea que propone Leibniz: Calculemus! [65]. Sorprendentemente este autor es uno de los pocos que ha hecho hincapié en los beneficios de esta función de los lenguajes formales. Lo cierto es que la idea general de los métodos algorítmicos fue uno de los componentes básicos de la tradición de la lógica algebraica, es decir, utilizar el álgebra simbólica para el estudio de la lógica. Sin embargo, en cierto sentido las funciones de expresividad y cálculo están en tensión entre sí, porque la razón principal de la primera parece ser la búsqueda de claridad, lo que a menudo es un obstáculo cuando se trata del cálculo eficiente. Como sugiere Krämer [66], la verdadera fuerza de los métodos de cálculo reside en la manipulación de signos, que se puede llevar a cabo incluso si el agente no sabe por qué son eficaces y correctos.

   Pero mejorar la claridad implica a menudo el uso de más símbolos, lo que afecta el rendimiento de los formalismos, porque son engorrosos para los fines de cálculo, aunque cognitivamente es más económico si se opera con un número menor de símbolos. Además, mientras más se piensa acerca de lo que se está haciendo cuando se trabaja con formalismos, es más probable que las intuiciones externas interfieran en el proceso, algo que se debe evitar en la mayoría de contextos. Por supuesto, el uso de un formalismo dado para hacerle frente a un problema particular se debe basar en cierto grado de justificación epistémica, que le permita adecuarse a esa aplicación. De hecho, cuando se desarrolla nuevos formalismos generalmente hay un proceso paralelo de búsqueda de justificación epistémica [67], pero una vez se alcanza un grado satisfactorio de certeza epistémica ya no es necesario que se recuerde cada vez, porque un formalismo dado de esta forma es digno de confianza.

En este tema hay que tener en cuenta que, desde un punto de vista epistémico, los lenguajes formales *son modelos* [68], es decir, idealizaciones simplificadas de fenómenos objetivos externos. Este punto de vista tiene la ventaja de reconocer que, casi inevitablemente, habrá huecos entre los modelos y los fenómenos, lo que representa una mejora respecto a la opinión de que el formalismo combina perfectamente los fenómenos, pero requiere una seria reflexión acerca de qué son realmente los lenguajes formales. Por otro lado, y según una creencia popular, son modelos de lenguas vernáculas/ordinarias, es decir, un modelo matemático de un lenguaje natural, más o menos en el mismo sentido de que, por ejemplo, una máquina de Turing es un modelo de cálculo, una colección de puntos de masa es un modelo de un sistema de objetos

físicos y que la construcción de Bohr es un modelo de un átomo. En otras palabras, un lenguaje formal muestra ciertas características de los lenguajes naturales, o idealizaciones de las mismas, mientras ignora o simplifica otras características [69].

De esta manera se puede argumentar que los lenguajes formales son modelos de lenguajes comunes en contextos específicos, matemáticos en particular, como sugiere Shapiro, lo que constituye una posición más plausible en el espíritu del proyecto expresivo de Frege [13] de diseñar un lenguaje para formular versiones claras, y sin espacios en blanco, para las pruebas matemáticas. Pero eso sería como restringir el ámbito de aplicación de estos lenguajes solamente a contextos matemáticos, y si bien esa era la intención original de Frege, el desarrollo de la lógica ha ido mucho más allá, por lo que sería conveniente considerar equivocada una definición en la que los lenguajes formales sean modelos de fenómenos lingüísticos.

El supuesto parece ser que un lenguaje formal, para describir un fenómeno objetivo determinado, hace un desvío a través del discurso ordinario sobre el fenómeno destino de que trate, por lo que un lenguaje formal sería muy similar a cualquier formalismo matemático, un lenguaje en sí mismo que puede caracterizar directamente el fenómeno destino sin la mediación de lenguajes ordinarios. Por ejemplo, el formalismo matemático de Maxwell en la teoría del electromagnetismo, porque él no tenía la intención de reglamentar el discurso ordinario sobre los fenómenos electromagnéticos, sino describir directamente y en sus propios términos el fenómeno en cuestión.

A raíz de esto se necesita aclarar que los lenguajes formales tienen sus orígenes en la tradición matemática de la notación algebraica, en lugar de en la tradición lógica reglada y esquemática. Observaciones que apoyan la afirmación de que el impacto cognitivo real de los lenguajes formales pertenece al reino del cálculo y la computación, en lugar de a la esfera de la expresividad. Uno de los resultados de este enfoque es la opinión de que los lenguajes formales no se deben ver como modelos de lenguajes comunes, teniendo en cuenta que sus limitaciones pasan por la imposibilidad de maximizar tanto la expresividad como la trazabilidad.

3. ANÁLISIS Y DISCUSIÓN

Anteriormente, a la lógica de la sintaxis de los lenguajes formales se le consideraba como un tema bastante aburrido, restringido a unos pocos párrafos apresurados sobre cuestiones de notación. Este indebido abandono llamó la atención desde los primeros días de la lógica en el siglo XX, cuando los académicos se enfrascaron en una formalización implacable [70]. El desánimo actual incluso se puede reforzar por el creciente interés en los lenguajes naturales, cuya estructura lógica es más viva y llama la atención en estos días, pero algunos pocos afirman que el estudio de la lógica de la sintaxis puede ser tan interesante como la lógica de la semántica.

Aunque los lenguajes formales son de contexto-libre, su complejidad parece aumentar una vez que se añaden condiciones formales, aparentemente inocuas, un fenómeno que se puede contrastar con las discusiones de la complejidad de los lenguajes naturales, donde las estimaciones tienden más a disminuir, desde un contexto libre, a uno regular [71].

Por otro lado, aparecen los fragmentos naturales de los lenguajes formales, en los que un formalismo, como la lógica de predicados, puede ser visto como el resultado de dejar que ciertas construcciones lingüísticas (negación, cuantificación, …) lleguen hasta el infinito. Algo que, en retrospectiva, se puede reducir hasta en jerarquías ascendentes de fragmentos de complejidad creciente, medidas, por ejemplo, por la profundidad de anidamiento de los cuantificadores.

Pero esto, que puede contar como un fragmento natural, es una noción altamente intencional que depende de las aplicaciones y los puntos de vista previstos. Aun así, hay que tener en cuenta que en la lógica de predicados es posible encontrar ejemplos lingüísticamente motivados [72], que generan nuevas preguntas acerca de lenguajes, supuesta y totalmente conocidos. Además, a la lógica de la semántica en sí no la afecta los cambios en la perspectiva sintáctica, en parte porque las estrategias de interpretación para los lenguajes formales utilizan las nociones de la Teoría de Autómatas para implementar las ideas acerca de la interpretación radical o progresiva; aunque resulta que esa interpretación es posible en los lenguajes regulares, donde el alcance de la desambiguación aún no tiene lugar. De hecho, la tesis general defendida por muchos es que la no-ambigüedad es un resultado, en lugar de una condición previa de la interpretación exitosa [73].

Por otro lado, algunos trabajos en la literatura de este tema se dan entorno a la complejidad de los teoremas lógicos y las pruebas [74], aunque se pueden ver como fragmentos de texto en lugar de afirmaciones individuales, si bien también se trabaja el tema de un adecuado formato sintáctico para hacer inferencia, donde las conclusiones es que no hay solamente un nivel donde la inferencia tiene lugar. Por lo tanto, se hace visible la necesidad de mayor investigación en este sentido. Asimismo, la sintaxis, la semántica y la lógica de los lenguajes formales a menudo se presentan de manera unitaria, lo que sugiere que, por ejemplo, la semántica del lenguaje natural debe crear un nivel de uso múltiple de las formas lógicas, donde se lleve a cabo la interpretación y la inferencia en el buen espíritu composicional, pero para la lógica eso es demasiado estricto.

También hay que tener en cuenta que el término *formal* es omnipresente en las discusiones filosóficas, particularmente en las relativas a la lógica, en las que, incluso con un examen superficial, se encuentra que parece utilizarse equívocamente y, a menudo, de formas imprecisas. Sin duda, esencialmente es un término general que abarca una amplia gama de conceptos y fenómenos relacionados, pero distintos, y como resultado cualquier análisis, donde se supone que el concepto desempeña un papel prominente, parece requerir un examen preliminar, particularmente una especificación de cuáles de sus variantes son relevantes para el contexto. En este sentido, MacFarlane [75] indaga acerca de qué significa decir que la lógica es formal, y con el fin de explorar esa naturaleza de la lógica presenta una taxonomía de los diferentes significados del término *formal*. Formula criterios que la delimitan sobre la base de la idea de lo formal y, mediante discusión, distingue diferentes variaciones que presenta en dos nociones: 1) formal como *asimilable*, y 2) formal como *legítimo*. Pero su taxonomía como un todo no puede ser adecuada totalmente en el contexto de las diferentes interpretaciones.

Mientras tanto, Novaes [76] presenta una taxonomía alternativa para las diferentes nociones de lo formal, principalmente (aunque no exclusivamente) en lo relacionado con la lógica, aunque su objetivo no es analizar la noción de lo formal como tal sin la presión de los objetivos externos. Esta taxonomía tiene cierta superposición con la de MacFarlane, pero en lugar de la dicotomía asimilable/legítimo propone dos grupos principales: 1) el *protocolar*, al que pertenecen las formas y que tiene cinco variaciones, y 2) el *formal*, al que pertenecen las reglas y que tiene tres variaciones. En relación con las formas lo protocolar se relaciona esencialmente con lo que cada cosa *es* y, como pertenecientes a las reglas, se relaciona esencialmente con lo que cada cosa *hace*. Por otra parte, el contrario del término formal, como referente a las reglas, generalmente es informal, mientras que como perteneciente a las formas, generalmente es material. Así que, claramente hay diferencias importantes entre los dos grupos, lo que sin duda debe ser tenido en cuenta en cualquier taxonomía.

Si bien hay que tener en cuenta que una cuestión clásica en el estudio de los lenguajes es poder medir la variación que se presenta entre sus diferentes géneros, el asunto inmediato que hay

resolver, cuando se estudia la estructura socio-lingüística, es la cuantificación de la dimensión de estilo [77], es decir, las variaciones de estilo del lenguaje. Esto se debe a que las personas se expresan de diferentes maneras, y a que individualmente expresan la misma idea de diversas maneras y de acuerdo con la audiencia a la que se dirigen, utilizando diferentes expresiones y diferentes estructuras gramaticales [78]. Ese número de posibles variaciones es tan amplio que el problema de Labov [77] parece irresoluble como un todo.

Esta cuestión se puede simplificar sustancialmente cuando el análisis se centra solamente en un aspecto o dimensión de estilo y, actualmente, el más utilizado es la *formalidad*. Por otro lado, aunque la mayoría de personas tiene al menos una idea y realiza distinción intuitiva entre una expresión formal y una informal, muy pocas tienen una definición clara y general para formalidad. La definición de Richards y Schmidt [79] para el discurso formal, como un tipo de alocución que se utiliza en situaciones en las que hay que ser cuidadoso con la pronunciación, la elección de las palabras y la estructuración de las oraciones dibuja la idea de una situación formal, pero no ofrece una definición para el discurso formal como tal, porque simplemente describe una hipótesis acerca de a qué se debe prestar atención al hablar en ciertos contextos. Por lo tanto, el criterio principal para definir la formalidad al hablar no es lingüístico. Lo mismo podría decirse de las descripciones de Labov [77] y Tarone [80] acerca de que la presencia de señales en el canal, es decir, las modulaciones que afectan el discurso en su conjunto (volumen, velocidad, tono, ritmo), indican un estilo informal, pero tampoco revelan nada acerca de la estructura intrínseca de la informalidad.

Por eso es que desde hace tiempo los lingüistas han tratado de determinar el nivel de formalidad de un lenguaje, considerado desde la frecuencia de las palabras hasta la semántica y las formas y estructuras gramaticales [81-84]. Sin embargo, esta manera de definir formalidad es empírica, intrínsecamente limitada, y muy dependiente de un lenguaje y una cultura específica. Entonces, esta ambigüedad alrededor de la definición y la falta de una buena y adecuada cuantificación de la dimensión de estilo, obstaculiza la investigación sociolingüística, tal como lo previó Labov. Pero la realidad es que los investigadores se dieron cuenta de que es difícil encontrar diferencias confiables y objetivas entre un discurso cuidadoso y uno casual [85-87].

Un supuesto incluido en la mayoría de enfoques es que un discurso formal se caracteriza por una atención especial a la forma, donde el orador trata de aproximarse lo más posible a un estándar de pronunciación. Pero este supuesto no encaja, porque primero habría que preguntarse por qué ese orador decide invertir tanta atención en la forma de sus oraciones. Aunque podría necesitar hacerlo en ocasiones importantes en las que se debe guardar apariencias y formas, porque la comunicación sigue siendo el propósito esencial del lenguaje, es decir, que se entienda el mensaje. Por eso, aunque el lenguaje parece tener una función puramente social y no-informativa, todavía comunica el mensaje elemental, y trata de hacerlo con la mayor claridad posible. P

or lo que algunos asumen que la función del lenguaje es obedecer las máximas generales de la conversación [88], entre las que se incluye requisitos de carácter informativo ciertos, relevantes, limpios y no-ambiguos. Por lo tanto, desde esta perspectiva el orador podría prestar mayor atención a la forma si el objetivo fuera asegurar que su discurso no fuera mal interpretado.

Pero esto es necesario únicamente en situaciones en las que se requiere una efectiva comunicación y, por lo tanto, es más difícil que en circunstancias ordinarias. Al final este análisis conduce necesariamente a dos tipos de formalidad: 1) *superficial*, caracterizada por una atención especial a la forma y directamente proporcional al significado de la palabra formal, como observación rigurosa, precisa, directa y ceremoniosa de las formas; y 2) *profunda*, que es explícita

y definitiva en oposición a lo que es materia de la comprensión tácita, es decir, atención a la forma en aras de alcanzar una comprensión inequívoca del significado preciso del discurso.

En todo esto, este trabajo se centra en analizar el nivel de la formalidad profunda de los lenguajes formales, debido a que como teoría es fundamental en la especificación del software y porque tiene aplicaciones prácticas más amplias que la formalidad superficial. Además, porque en la mayoría de casos la atención a la forma a nivel superficial solamente refleja una expresión irrefutable del nivel de profundidad. Este tipo de análisis resulta de un discurso mal intencionado, por ejemplo, el orador puede utilizar un estilo formal para dar la impresión de que presenta información objetiva y precisa, aunque realmente lo que quiere es ocultar detalles comprometedores. En casos como este, donde la formalidad simplemente es convencional, es prioritario mantener la forma inicial sobre el mensaje original, donde el significado literal de las expresiones se pierde o es ambiguo.

En este sentido, aunque el formalismo puede parecer puramente superficial, a menudo hay un mensaje profundamente formal en las interpretaciones de segundo orden, pero en este trabajo no se discute esta situación ya que es mucho más compleja y menos común que el caso general de la formalidad profunda. Otra ventaja de realizar este tipo de análisis es que las estructuras necesarias son más universales y menos específicas a un lenguaje formal, por lo que es más probable encontrar resultados de mayor utilidad para el desarrollo de software. Aunque la definición de profundidad puede parecer abstracta o teórica, aquí se demuestra que se puede operacionalizar más fácilmente, debido a que la métrica resultante distingue con eficacia el nivel de formalidad de los lenguajes formales, lo que en sí se puede considerar intuitivamente formal.

Esta caracterización de formalidad, orientada a evitar la ambigüedad, se relaciona directamente con el significado del término *formal* en matemáticas y lógica, lo que es beneficioso para el logro de los objetivos de este capítulo. Además, es común que los lenguajes naturales como el español, sean muy diferentes de formalismos matemáticos como el cálculo proposicional, a pesar de que, aparentemente, comparten términos y conceptos [89]. Pero la informalidad de estos lenguajes es lo que genera ambigüedad en la elicitación de requisitos, algo que intentan disipar los lenguajes formales. Aun así, las afirmaciones en lenguaje natural pueden parecer claras e inconfundibles, pero el hecho de que sean personas las encargadas de especificar las necesidades de un sistema genera una libertad de interpretación que ocasiona pérdida de tiempo y dinero. En realidad, lo que realmente diferencia a los lenguajes formales es que tratan de lograr la misma claridad, pero sin supuestos no-declarados.

Por eso es que la especificación formal, al distanciarse del contexto inmediato de las necesidades del cliente, es menos directa que la informal, porque para expresar los significados hace uso de las características más esenciales del contexto. Por su parte, la especificación informal es más interactiva e involucra mayoritariamente a las reacciones de las partes interesadas, los eventos y otros elementos del contexto, en lugar de describir las necesidades desde un punto de vista impersonal individual, una cuestión que genera libre interpretación y documentación. La conclusión entonces es que en la especificación el nivel de formalidad de los lenguajes naturales depende de cada sistema, pero seguirá siendo un elemento subjetivo que obedece a si las partes interesadas prefieren precisión sobre inmediatez e individualismo sobre participación, por lo que sus interpretaciones tendrán una carga cognitiva adicional. En estos casos la forma comúnmente utilizada para establecer los requisitos es por observación empírica, para luego comparar su formalidad general con la esperanza de encontrar recurrencia en las expresiones originadas desde diferentes interpretaciones y por diferentes sujetos individuales.

4. CONCLUSIONES

En este capítulo se explora las nociones de la lógica y la formalidad desde un punto de vista relevante para el concepto de lenguajes formales, que algunos investigadores denominan de-semantificación de lo formal como computable, al mismo tiempo que de las implicaciones cognitivas. Pero el objetivo de este trabajo es argumentar que ambas nociones son históricamente ricas, filosóficamente refinadas y completamente integradas en los debates filosóficos de las matemáticas y en la noción de la computabilidad.

El resultado principal es que la integración de la filosofía y de las perspectivas cognitivas acerca de la lógica de los lenguajes formales es un desarrollo natural, dadas las similitudes subyacentes con el principio de la tecnología de los Métodos Formales. Por lo tanto, y de acuerdo con la conceptualización que se presenta en esta investigación, las características más importantes de un lenguaje formal son: es escrito, su dimensión semántica no es representacional sino basada en el uso de los agentes, su función es predominantemente operativa en lugar de expresiva, y es formal porque es el producto de un proceso de de-semantificación y muestra propiedades computacionales clave, que a su vez depende crucialmente de su condición como artefacto cognitivo externo.

Para el surgimiento de esta re-conceptualización de los lenguajes formales se escrudiñó una serie de premisas y supuestos, y se hizo una re-evaluación a la relación lógica-formalidad; además, se tuvo en cuenta la adopción de un enfoque lenguaje-como-práctica, la ampliación de la noción de significado para incluir lo que se ha descrito como sentido intervencionista y la interrupción de la dicotomía natural vs artificial con respecto a los lenguajes. Los autores son conscientes del carácter un tanto peculiar de esta concepción de los lenguajes formales, además del hecho de que no parece reflejar gran parte de las prácticas actuales de la lógica. Pero actualmente los lógicos no parecen utilizar lenguajes formales para razonar con, sino predominantemente para, razonar sobre. Por eso se confía en la voluntad del lector para aceptar esta hipótesis de trabajo y se espera recompensarlo en próximas publicaciones, en las que se presentará reflexiones metodológicas más generales acerca del uso de los Métodos Formales, pero sobre la base de los lenguajes formales como tecnología cognitiva.

Esta es claramente una investigación sobre la metodología, más específicamente en las entradas y salidas de los Métodos Formales, en la lógica, la filosofía y en otros contextos. Tal vez, irónicamente, los Métodos Formales no se discuten ampliamente, pero en realidad no se emplean aquí porque es una investigación sobre fórmulas lógicas y formales con muy pocas fórmulas lógicas y formales. En lugar de ello la metodología adoptada es lo que se podría describir como integradora, combinando el análisis filosófico tradicional con la atención a elementos empíricos e históricos a la vez. Si bien el análisis filosófico y los elementos empíricos a menudo se combinan cada vez más en la filosofía dominante, y si bien el papel del análisis histórico para la teorización filosófica tiene mucho reconocimiento, hasta el momento la combinación de los tres enfoques es inusual. Una excepción podría ser el proyecto de Netz [54] sobre la historia cognitiva que, sin embargo, se mantiene esencialmente a nivel programático.

Por otro lado, los Métodos Formales en filosofía corresponden a la aplicación de herramientas matemáticas y lógicas para investigar cuestiones filosóficas, por ejemplo, en el desarrollo de mundos semánticos posibles para el análisis de los conceptos de necesidad y posibilidad, aplicaciones del marco bayesiano a los problemas en la epistemología (dando lugar a la llamada epistemología formal), la explicación carnapiana, y muchos otros.

En cierto sentido, en la medida en que la lógica sustenta investigaciones filosóficas teóricas (que en realidad es un punto discutible), y mientras que la lógica en general proporcione metodologías generales para la investigación filosófica, los Métodos Formales tendrán una larga y distinguida historia en la práctica filosófica. Pero cuando se habla de ellos en filosofía se refiere específicamente a la aplicación de la lógica simbólica y matemática a los problemas filosóficos, que cada persona tiene en mente. Naturalmente, los lenguajes formales comparten el carácter *a priori* de esta categoría, pero la diferencia fundamental es el lenguaje en el que se lleva a cabo la investigación, es decir, semi-regimentado, pero esencialmente lenguaje cotidiano vs lenguaje formal de la lógica y las matemáticas y, tal como se ha argumentado en todo el contenido, *esta diferencia hace toda la diferencia*.

REFERENCIAS

[1] Goguen J. (1999). Hidden algebra for software engineering. Combinatorics, Computation and Logic 21, 35-59.

[2] Polansky J. y Sinclair M. (2014). The importance of training in Formal Methods in Software Engineering. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 4(2), 52-56.

[3] Rushby J. (1995). Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1. SRI International.

[4] Woodcock J. et al. (2009). Formal methods: Practice and experience. ACM Computing Surveys 41(4), 1-36.

[5] Anya P. y Smith G. (2014). Qualitative research methods in Software Engineering. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 4(2), 14-18.

[6] Pole, K. (2009). Diseño de metodologías mixtas. Una revisión de las estrategias para combinar metodologías cuantitativas y cualitativas. Renglones 60, 37-42.

[7] Pereira Z. (2011). Mixed Method Designs in Education Research: A Particular Experience. Revista Electrónica Educare 15(1), 15-29.

[8] Serna E. (2018). Metodología de investigación aplicada. En E. Serna (Ed.), Ingeniería: Realidad de una disciplina (pp. 6-33). Editorial Instituto Antioqueño de Investigación.

[9] Kowalski R. (1979). Logic for problem solving. North-Holland.

[10] Kowalski R. (1986). The limitations of logic. En C. Thanos y J. Schmidt (Eds.), Knowledge Base Management Systems (pp. 477-489). Springer.

[11] Kowalski R. (2010). Computational logic and human thinking: How to be Artificially Intelligent. Cambridge University Press.

[12] Kowalski R. y Sadri F. (2014). A logical characterization of a reactive system language. Lecture Notes in Computer Science 8620, 22-36.

[13] Frege G. (1879). Begriffsschrift: Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Verlag.

[14] Frege G. (1893). Grundgesetze der Arithmetik, Begriffsschriftlich abgeleitet. Verlag.

[15] Gödel K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. Monatshefte für Mathematik und Physik 38, 173-198.

[16] Russell B. y Whitehead A. (1913). Principia mathematica. Cambridge University Press.

[17] Hilbert D. (1902). Grundlagen der geometrie. Verlag.

[18] Robinson J. (1965). A machine-oriented logic based on the resolution principle. Journal for the Association for Computing Machinery 12(1), 23-41.

[19] Gillies D. (1996). Artificial Intelligence and scientific method. Oxford University Press.

[20] Muggleton S. (1992). Inductive logic programming. Academic Press.

[21] Church A. (1932). A set of postulates for the foundation of logic. Annals of Mathematics 33(2), 346-366.

[22] Kleene S. y Rosser J. (1935). The inconsistency of certain formal logics. Annals of Mathematics 36(3), 630-636.

[23] Priestley M. (2011). A science of operations: Machines, logic and the invention of programming. Springer.

[24] Frege G. (1967). Begriffsschrift: A formula language, modeled upon that of arithmetic, for pure thought. En J. van Heijenoort (Ed.), From Frege to Gödel: A source book in mathematical logic 1879-1931 (pp. 5-55). Harvard University Press.
[25] Davis M. (1995). Influences of mathematical logic on computer science. En R. Herken (Ed.), The Universal Turing Machine (pp. 289-299). Springer.
[26] Frege G. (1884). Die Grundlagen der Arithmetik: Eine logisch mathematische untersuchung über den begriff der zahl. Verlag.
[27] de Saussure F. (1916). Cours de linguistique générale. Payot.
[28] Chomsky N. (2012). Le langage et la pensé. Payot.
[29] van Orman W. (1953). From a logical point of view. Harper Torchbooks.
[30] Davidson D. (1967). Truth and meaning. Synthese 17(3), 304-323.
[31] Locke J. (1690). An essay concerning human understanding. Eliz. Holt.
[32] Berkeley G. (1710). Treatise concerning the principles of human knowledge. Thomas.
[33] Hume D. (1737). A Treatise of human nature. The University of Adelaide Press.
[34] Wittgenstein L. (1922). Logisch-philosophische abhandlung. Kegan Paul.
[35] Burge T. (1990). Frege on sense and linguistic meaning. En D. Bell y N. Cooper (Eds.), The analytic tradition, meaning, thought and knowledge (pp. 242-270). Blackwell.
[36] Kripke S. (1982). Wittgenstein on rules and private language: An elementary exposition. Harvard University Press.
[37] Peirce C. (1873). Description of a notation for the logic of relatives: Resulting from an amplification of the conceptions of Boole's calculus of logic. Memoirs of the American Academy of Arts and Sciences New Series 9(2), 317-378.
[38] Brentano F. (1995). Descriptive psychology. Routledge.
[39] Searle J. (1969). Speech acts: An essay in the philosophy of language. Cambridge University Press.
[40] Mill J. (1843). System of logic: Raciocinative and Inductive. Holt.
[41] Sapir E. (1921). Language: An introduction to the study of speech. Harcourt.
[42] Whorf B. (1940). Science and linguistics. MIT Technology Review 42, 229-231.
[43] Dummett M. (1993). The Seas of Language. Oxford University Press.
[44] Grice P. (1989). Studies in the way of words. Harvard University Press.
[45] Fodor J. (2000). The mind doesn't work that way: The scope and limits of computational psychology. MIT Press.
[46] Tarski A. (1944). The semantic conception of truth and the foundations of semantics. Philosophy and Phenomenological Research 4(3), 341-376.
[47] Carnap R. (1928). The logical structure of the world: Pseudoproblems in philosophy. University of California Press.
[48] Montague R. (1980). Logic: Techniques of formal reasoning. Oxford University Press.
[49] Davidson D. (1972). Semantics of natural languages. Springer.
[50] Grice H. (1975). Logic and conversation. En P. Cole y J. Morgan (Eds.), Syntax and Semantics 3 (pp. 22-40). Academic Press.
[51] Ryle G. (1951). The concept of mind. Hutchinsons University Library.
[52] Strawson P. (1952). Introduction to logical theory. Methuen.
[53] Staal F. (2006). Artificial languages across sciences and civilizations. Journal of Indian Philosophy 34(1-2), 89-141.
[54] Netz R. (1999). The shaping of deduction in Greek mathematics: A study in cognitive history. Cambridge University Press.
[55] Staal F. (2007). Artificial languages between innate faculties. Journal of Indian Philosophy 35(5-6), 577-596.
[56] Thue A. (1906). Uber unendliche Zeichenreihen. Kristiania.
[57] Thue A. (1912). Uber die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Germany.
[58] Post E. (1936). Finite combinatory processes-formulation. Journal of Symbolic Logic 1(3), 103-105.
[59] Turing A. (1936). On computable numbers with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42(1), 230-265.
[60] Chomsky N. (1956). Three models for the description of language. Transactions on Information Theory 2, 113-124.

[61] Lindenmayer A. (1968). Mathematical models for cellular interactions in development. Journal of Theoretical Biology 18(3), 280-299.
[62] Stokhof M. (2007). Hand or hammer? On formal and natural languages in semantics. Journal of Indian Philosophy 35(5-6), 597-626.
[63] Carnap R. (1934). The logical syntax of language. Open Court.
[64] Leibniz G. (). Logical papers: A selection. Clarendon Press.
[65] Lenzen W. (2004). Leibniz's logic'. En D. Gabbay y J. Woods (Eds.), Handbook of the history of logic Vol. 3: The Rise of Modern Logic: From Leibniz to Frege (pp. 1-83). Elsevier.
[66] Krämer S. y Writing S. (2003). Notational iconicity, calculus: On Writing as a cultural technique. Modern Languages Notes 118(3), 518-537.
[67] Heeffer A. (2007). Humanist repudiation of eastern influences in early modern mathematics. Ghent University.
[68] Frigg R. y Hartmann S. (2006). Models in science. En Z. Edward y Pfeifer J. (Eds.), The Stanford Encyclopedia of Philosophy (pp. 740-749). Stanford University Press.
[69] Shapiro S. (1998). Logical consequence: Models and modality. En M. Schirn (Ed.), Philosophy of Mathematics Today (pp. 131-156). Clarendon Press.
[70] Bacon J. (1985). The completeness of a predicate-functor logic. Journal of Symbolic Logic 50, 903-926.
[71] Suppes P. (1976). Elimination of quantifiers in the semantics of natural language by use of extended relational algebras. Revue Internationale de Philosophie 117(8), 243-259.
[72] Kamp H. (1984). A theory of truth and semantic representation. En J. Groenendijk et al. (Eds.), Truth, Interpretation and Information (pp. 1-41). Foris.
[73] Durnmett M. (1973). Frege - The Philosophy of Language. Duckworth.
[74] van Benthem J. (2000). Meaning: Interpretation and Inference. Synthese 73(3), 451-470.
[75] MacFarlane J. (2000). What does it mean to say that logic is formal? Doctoral dissertation. University of Pittsburgh.
[76] Novaes D. (2011). Medieval theories of supposition. En H. Lagerlund (Ed.), Encyclopedia of Medieval Philosophy (pp. 1229-1236). Springer.
[77] Labov W. (1972). Sociolinguistic patterns. University of Philadelphia Press.
[78] Bell A. (1984). Language style as audience design. Language in Society 13(2), 145-204.
[79] Richards J. y Schmidt R. (2002). Dictionary of language teaching and applied linguistics. Pearson.
[80] Tarone E. (1988). Variation in interlanguage. Edward Arnold.
[81] Blanche C. (1991). Le français parlé: Etudes gramaticales. Editions du CNRS.
[82] Chierchia G. y McConnell S. (2000). Meaning and grammar: An introduction to semantics. MIT Press.
[83] Halliday M. y Webster J. (2006). On language and linguistics. Bloomsbury Academic.
[84] Akmajian A. et al. (2010). Linguistics. MIT Press.
[85] Rickford J. y McNair F. (1995). Adressee -and topic- influenced style shift. A quantitative sociolinguistic study. En D. Bier & E. Finegan (Eds.), Sociolinguistic perspectives on register variation (pp. 235-276). Oxford University Press.
[86] Barbara S. (2003). Controversies in applied linguistics. Oxford University Press.
[87] Daniela I. y Reiss C. (2013). I-language: An introduction to linguistics as cognitive science. Oxford University Press.
[88] Alduais M. (2012). Conversational implicature (flouting the maxims): Applying conversational maxims on examples taken from non-standard Arabic language, Yemeni dialect, an idiolect spoken at IBB city. Journal of Sociological Research 3(2), 376-387.
[89] Toledo J. (2013). Introduction of the Formal Methods in Software Engineering training. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 3(1), 25-32.

CAPÍTULO V
La lógica en las Ciencias Computacionales[1]

Edgar Serna M.
Instituto Antioqueño de Investigación

En este capítulo se presenta un análisis a la necesidad de incluir a la lógica en los procesos formativos en Ciencias Computacionales CC. Se parte de un recorrido a la historia de la lógica en estas ciencias, posteriormente se describe la relación y la necesidad de incluirla en los procesos formativos relacionados, y al final se analiza qué, cuándo y qué tan profundo se debería trabajar en la formación en CC. Se trata de una revisión al estado de esta cuestión y a la importancia de incluir esta temática en los planes de estudios de pregrados y posgrados relacionados con estas Ciencias y con las Tecnologías de la Información TI. Los resultados tienen amplio impacto en el desarrollo de las Ciencias Computacionales, y se reflejarán en los procesos formativos relacionados al igual que en el papel cada vez mayor de la lógica en el desarrollo profesional.

[1] Publicado en la Revista de Educación en Ingeniería 8(15), 62-68. 2013.

INTRODUCCIÓN

En los últimos años se ha desarrollado poderosas herramientas para verificar especificaciones formales de sistemas hardware y software; la industria de TI se ha dado cuenta del impacto y la importancia que tienen en sus propios procesos de diseño e implementación, y diversas empresas las investigan e incorporan en sus departamentos de planeación y producción. Para trabajar en estos procesos se requiere una formación formal básica que les permita a estudiantes y profesionales desarrollar habilidades para utilizar, *razonar* y potencializar los sistemas.

El cambio de las TI para favorecer al acceso de datos con base en Internet y el procesamiento, también genera un incremento en la demanda por profesionales calificados que puedan razonar acerca del software sofisticado basado en agentes autónomos y capaces de interactuar con otros agentes para recopilar la información necesaria en las grandes redes.

Para aportar al hecho de que el trabajo en Ciencias Computacionales requiere la aplicación y el manejo adecuado de la lógica como componente formal, en este capítulo se presenta un análisis a esas necesidades formativas, y se analiza las bases de la formación en lógica que requieren los estudiantes y profesionales para desempeñarse en este campo laboral. Además, se hace una introducción a los marcos lógicos utilizados para modelar y razonar acerca de los sistemas informáticos. El objetivo es proporcionar un contenido que se pueda utilizar para diseñar cursos en lógica computacional, como una contribución que permita adaptarlos rápidamente a los actuales entornos profesionales, en medio del acelerado y cambiante entorno de las TI.

1. LA LÓGICA EN LAS CIENCIAS COMPUTACIONALES

La Magna Charta Universitatum [1] emitida con ocasión de los 900 años de la fundación de la Universidad de Bolonia, resumió la misión de las universidades en la sociedad moderna. Ese mismo año, y por iniciativa del ministro francés de educación, la Sorbonne Declaration [2] estableció el reconocimiento mutuo de las respectivas titulaciones, como un objetivo común de los cuatro países signatarios: Francia, Alemania, Italia y Reino Unido. Otros países aceptaron esas ideas y expresaron su disposición de unirse al proyecto.

Para enfatizar su compromiso, los ministros de educación de 29 países se reunieron en Bolonia en 1999 y se comprometieron, en una declaración conjunta [3], a establecer un espacio europeo en educación superior. El objetivo del Bologna Process era lograr que los sistemas europeos de formación superior confluyeran hacia un sistema más transparente, mediante el cual los diferentes sistemas nacionales utilizaran un marco común basado en tres ciclos formativos: Licenciatura, Maestría y Doctorado. Desde entonces, los países signatarios han hecho revisiones constantes a los objetivos del proceso y para clarificar algunas cuestiones en su implementación.

Posteriormente, aunque de manera tangencial, otros países en el mundo entraron al proyecto con el objetivo de adoptar los principios que en él se discuten e implementarlos en sus propios sistemas de educación. En América Latina se está trabajando seriamente al respecto, y algunos países han comenzado a modificar sus sistemas en pregrado y posgrado; la idea es no quedarse rezagados con respecto a los progresos que se hacen en otros continentes, porque las TI requieren actualización constante y profesionales capacitados para su explotación. Por otro lado, la revisión y re-construcción de los planes de estudio en Ciencias Computacionales se realiza bajo la supervisión de investigadores interesados en el tema, y se ha llegado a conclusiones como que es necesario contar con una profesionalización en desarrollo de software, es decir, una Ingeniería del Software cuyos productos sean reconocidos por su calidad y fiabilidad (Serna, 2011, 2011a).

Aunque esta área se considera actualmente como sucesora directa de la Ingeniería de Sistemas y los planes de estudio desarrollados están fuertemente basados en las experiencias locales relacionadas con el mantenimiento de más de 30 años de la misma, los cambios en el sistema formativo siempre brindan la oportunidad de hacer una revisión importante de los objetivos y del contenido en ambas áreas de formación. La revisión que se presenta en este trabajo se llevó a cabo teniendo en cuenta el papel y el contenido de la lógica en la formación en Ciencias Computacionales, especialmente en el área de la Ingeniería del Software.

A comienzos de siglo pasado, Hilbert [6] consideraba a la lógica como una teoría axiomatizada. Según este enfoque es posible demostrar teoremas por medio de los métodos matemáticos tradicionales, sin embargo, no existía algoritmos que soportaran la construcción de tales deducciones. El primer avance significativo en este sentido se debe a Gentzen [7], con el desarrollo de la técnica natural de deducción y el cálculo sucesivo, con los que creó una caja de herramientas sintácticas especiales para probar teoremas automáticamente. Por su parte, Herbrand [8] aportó la probabilidad de insatisfacción sobre los universos de Herbrand mediante un modelo teórico, e hizo posible la re-escritura del problema de decisión de primer orden como una fórmula proposicional que permitía expandirlo sobre dichos universos, con lo que el problema de demostrar un teorema se redujo a la revisión de las fórmulas a través de una caja de herramientas de lógica proposicional.

En los años 50, cuando los computadores fueron accesibles, Davis y Putnam [9] utilizaron los resultados de Herbrand y elaboraron el primer algoritmo de computador para demostrar el teorema. Newell y Simon [10] desarrollaron el General Problem Solver, y Newell, Shaw y Simon [11] diseñaron el sistema Logic Theorist, con los que impactaron la Inteligencia Artificial contemporánea. El primero utiliza el algoritmo del British Museum, un método de búsqueda horizontal a ciegas de bajo rendimiento, con base en los axiomas y reglas de inferencia dadas por Russell y Whitehead [12].

La investigación del problema de insatisfacción de las fórmulas Conjunctive Normal Form CNF canónicas y la generalización de la regla de resolución proposicional para las fórmulas de primer orden, dio origen a la resolución lógica proposicional o resolución básica. Robinson [13] definió la noción de la resolución de primer orden con base en que, si la resolución existe, es posible utilizar dos cláusulas que contengan las instancias básicas, cuando aparece una pareja literal básica complementaria. Además, reconoció que poder unificar el par literal original es la condición para que una resolución básica exista, principio con el que creó el cálculo de resolución de primer orden. Posteriormente, otros trataron de desarrollar estrategias de resolución para simplificar la implementación, pero hasta el momento las principales estrategias resultantes son la resolución semántica y la lineal (cálculo completo), y la resolución de entrada lineal y la resolución de unidad, que son posibles de implementar, pero que actualmente son cálculo incompleto.

En los años 60 surgió una demanda por la aplicación de la lógica en conexión con el análisis y la síntesis, lo que significó que las propiedades de un programa se pudieran describir mediante fórmulas lógicas (axiomas), y que fuera posible tratar de responder las preguntas acerca del funcionamiento correcto del mismo. La base formal de este enfoque fue principalmente la lógica de Hoare [14], cuyos aportes contienen los resultados más importantes en esta área [15-17]. Los llamados sistemas pregunta/respuesta fueron herramientas útiles, en los que la respuesta era una supuesta consecuencia/inferencia de las fórmulas que describen las propiedades del programa. Estos sistemas fueron dotados de una técnica especial que adicionalmente podía generar cierto tipo de respuestas acerca de la consistencia del teorema, siempre que se hubiera demostrado a sí mismo. En este proceso se utilizó la lógica de segundo orden, e incluso la

temporal, en la formalización [16, 18]. Los trabajos de Cliff [19], Hoare y Shepherdson [20], Loeckx y Sieber [21] y Goos et al. [22], ilustran que ésta es un área importante en el desarrollo y aplicación de la lógica.

Posteriormente, y utilizando la estrategia de entrada lineal incompleta, Colmerauer [23] y Kowalski [24] desarrollaron el teorema de pruebas del Prolog para las cláusulas de Horn de primer orden, que se utilizaban para definir las declaraciones lógicas de un programa. La investigación acerca de las capacidades de este sistema concluyó que, sobre la base del principio del modelo dado por los átomos de la primera capa, era posible desarrollar la noción del modelo mínimo de Herbrand [8] y de cierto tratamiento de negación.

Debido a que el resultado de un trazo de asignación, ordenado y definido en un reticulado completo, siempre tiene un punto fijo, entonces una de las interpretaciones de Herbrand es un subconjunto del conjunto de átomos de la primera capa sobre su universo, es decir, un sub-conjunto de la base de Herbrand, donde el conjunto de todas sus interpretaciones es un super-conjunto de toda la base.

El resultado del conjunto de todas las interpretaciones es un reticulado completo, con dos operadores y una relación del sub-conjunto. Una consecuencia directa de la función asignada a la lógica de un programa sobre esta red es la preservación de la ordenación, por lo que al menos tiene un punto fijo; además, se ha demostrado como el modelo mínimo de Herbrand de la lógica de un programa, con lo que se definió la extensión del punto fijo de la lógica de primer orden [25]), el cual juega un papel significativo en el enfoque DATALOG de las bases de datos relacionales y las bases del conocimiento.

La investigación de los sistemas de deducción lógica tuvo amplio impacto en el desarrollo de la teoría de modelos, principalmente en la evaluación de las cuestiones de la axiomatización [26]. La aparición de los lenguajes de programación funcionales, y el desarrollo de las tecnologías de programación paralela y concurrente, condujeron a la introducción de nuevas herramientas en la teoría de la programación. Una de ellas es λ-cálculo, presentado por Church [27] y utilizado para el tratamiento unificado de lógicas de orden diferente (nulo, primero, superior) [28]. La evolución de los lenguajes de programación involucró la evolución de varias herramientas, como π-cálculo y μ-cálculo, entre otras.

Desde los años 60, las cajas de herramientas de Inteligencia Artificial, la teoría de la programación y la teoría basada en el conocimiento, incluyen como herramientas lógicas no-clásicas a las diversas versiones de la lógica temporal, la lógica modal, las lógicas de valores diversos, las lógicas relevantes, la lógica no-monótona y la lógica difusa.

2. LA LÓGICA EN LA EDUCACIÓN EN CIENCIAS COMPUTACIONALES

Inicialmente se formaba en lógica, pero vista solo como una técnica descriptiva. Para los años 60 sus aplicaciones suponían el conocimiento de amplias áreas de la misma, lo que implicaba que en los planes de estudio de los cursos avanzados se debía incluir los temas básicos de la lógica matemática. Desde entonces la lógica se convirtió en una herramienta para otros campos de la informática, como la Inteligencia Artificial, la teoría de bases de datos relacionales y el análisis y síntesis de programas. A finales de esta década las bases de la lógica comenzaron a hacer parte de la formación en los primeros años de las CC, y las lecturas eran obligatorias debido a que estos campos del conocimiento comenzaron a jugar un papel importante en las aplicaciones; posteriormente se desplazaron a años superiores.

Posteriormente, se comenzaron a publicar trabajos para estudiantes de informática y para especialistas avanzados acerca de la lógica [29-37], al mismo tiempo que se editaba algunos manuales de resúmenes [38-44].

2.1 Qué, cuándo, cuánta lógica

En la fase actual del desarrollo de las Ciencias Computacionales se conoce cuál es el conocimiento y qué parte de la lógica es el que debe desarrollar un graduado, y se concluye que se le debe capacitar en lógica proposicional y de primer orden, a través de un enfoque orientado a: el lenguaje, el alfabeto, la sintaxis, la semántica, las reglas de re-escritura, la noción de consecuencia semántica, el teorema de demostración de problemas, el problema de la decisión, la solución semántica de la prueba de teoremas, el tratamiento sintáctico de la lógica y el principio de los sistemas de deducción. Además, que se debe incluir el cálculo de resoluciones y el método arbitrario, conectados por el sistema de Hilbert [45], como el más aceptado de los dos sistemas de deducción conocidos. También se debe abarcar las nociones de correctitud y completitud, de tal manera que se siente las bases necesarias para la introducción de: la lógica temporal de Hoare, la lógica de unidad, la lógica de punto fijo y la lógica descriptiva [46], utilizadas en la educación en diversas áreas de las Ciencias Computacionales.

En cierta medida también se puede utilizar en la formación en bases de datos, la teoría de la programación y la Inteligencia Artificial, y además son un punto de partida para introducir las lógicas no-clásicas [47]. Después de esta fundamentación se podrá introducir los lenguajes de programación funcionales [48]. Los cursos de pregrado enfocados en la teoría de base de datos o la Inteligencia Artificial no son los únicos que requieren el conocimiento de la lógica fundamental, también algunos cursos de posgrado se deben fundamentar en ella. Por eso, luego de analizar los currículos relevantes de algunas universidades en las que se imparte la lógica en la formación en posgrados, se concluye que los campos más importantes de formación en esta área son:

- Los fundamentos teóricos y los métodos de prueba de teoremas.
- Los problemas de formalización y axiomatización: problemas de las teorías axiomatizadas.
- La programación lógica y sus fundamentos teóricos: lógica de punto fijo y el modelo Herbrand.
- Cálculo en el marco de la programación funcional.
- Lógicas no-clásicas: temporal, polivalente, difusa, descriptiva, entre otras.

Estos temas son importantes para incluirlos en el marco de un programa de posgrado, al tiempo que se justifica la existencia de temas como lógica no-clásica a nivel de pregrado, al menos como curso electivo.

3. CONCLUSIONES

En este trabajo se presenta un breve recorrido histórico acerca de la lógica y las posibilidades y ventajas de la educación en Ciencias Computacionales con base en ella. Estas ideas parten de la conceptualización que introdujo el Tratado de Bolonia para la formación en informática, pero que redunda en todo el planeta. El acercamiento propuesto aquí hace hincapié en la formación acerca de los principales métodos de inferencia, incluyendo la resolución no causal, y ofrece una introducción a la teoría y aplicación de las lógicas de muchos valores, y de otras que se difunden y aplican en la industria de TI.

La formación en lógica se debe realizar no solo en el marco de las matemáticas, sino también en el de las Ciencias Computacionales, con un aspecto transdisciplinar para lograr la eficiencia, lo que

significa que ambas áreas del conocimiento tienen que ser colaborativas en el currículo [49]. Lo básico debe ser inherente a la formación lógico-matemática, como la abstracción, los lenguajes, las semánticas, los conectores, las interpretaciones, las tautologías, el cálculo proposicional y la lógica computacional. Pero en los posgrados se debe incluir aspectos avanzados, como sintaxis vs semánticas, representación de conocimiento, tareas problemáticas de la *vida real*, deducción y métodos formales, resoluciones y lógicas de muchos valores [50].

Además, se debe utilizar las herramientas informáticas actuales para que la formación sea más interesante para los estudiantes, como la lógica de programación, las máquinas de inferencia visual con ejemplos preparados, y las herramientas CASE, entre otras. También es necesario mantener una actualización constante mediante seminarios y asignaturas de aplicación práctica en la industria.

REFERENCIAS

[1] Bolonia. (1988). Magna Charta Universitatum. Recupertado: http://www.magna-charta.org /cms/cmspage.aspx?pageUid={d4bd2cba-e26b-499e-80d5-b7a2973d5d97}

[2] Sorbonne. (1988). Sorbonne Declaration. Recuperado: http://www.bologna-bergen2005.no/Docs/00-Main_doc/980525SORBONNE_DECLARATION.PDF

[3] CRE. (2000). The Bologna Declaration on the European space for higher education: An explanation. Recuperado http://ec.europa.eu/education/policies/educ/bologna/bologna.pdf

[4] Serna E. (2011). Systems engineering for the XXI century: A proposal from the academy. En Ninth Latin American and Caribbean Conference. Medellín, Antioquia.

[5] Serna E. (2011). Software Engineering is Engineering. Revista RACCIS 1(1), 34-43.

[6] Hilbert D. (1920). The Grounding of Elementary Number Theory, en From Brouwer to Hilbert: The debate on the foundations of mathematics in the 1920's de Mancosu. Oxford University Press.

[7] Gentzen G. (1934). Untersuchung Äuber das logische Schliessen. Mathematische Zeitschrift 39, 176-210.

[8] Herbrand J. (1968). Logical writings. Reidel Publishing.

[9] Davis M. y Putnam H. (1960). A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201-215.

[10] Newell A. y Simon H. (1956). The logic theory machine: A complex information processing system. IRE Transactions on Information Theory 2(3), 61-79.

[11] Newell A., Shaw J. y Simon H. (1959). Report on a general problem-solving program. En International Conference on Information Processing. Paris, France.

[12] Russell B. y Whitehead, A. (1997). Principia Mathematica to *56. Cambridge University Press.

[13] Robinson J. (1965). A machine-oriented logic based on the resolution principle. Journal of the ACM 12(1), 23-41.

[14] Hoare C.A.R. (1969). An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576-580.

[15] Manna Z. y Waldinger R. (1971). Toward Automatic Program Synthesis. Communications of the ACM 14(3), 151-165.

[16] Manna Z. y Pnueli A. (1974). Axiomatic approach to total correctness of programs. Acta Informática 3(3), 243-264.

[17] Pnueli A. (1977). The temporal logic of programs. En 18th Annual Symposium on Foundations of Computer Science. Rhode Island, USA.

[18] Kröger F. (1987). Temporal Logic of Programs. Springer.

[19] Cliff B. (1980). Software development. A rigorous Approach. Prentice Hall.

[20] Hoare C. y Shepherdson J. (1985). Mathematical Logic and Programming Languages. Prentice-Hall.

[21] Loeckx J. y Sieber K. (1987). The Foundations of Program Verification. Wiley.

[22] Goos G. et al. (2003). Verification: Theory and Practice. Springer.

[23] Colmerauer A. (1970). Les systèmes-q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. l'Université de Montréal.

[24] Kowalski R. (1979). Algorithm = Logic + Control. Communications of the ACM 22(7), 424-436.
[25] Abiteboul S. et al. (1995). Foundations of Databases. Addison-Wesley.
[26] Ebbinghaus H. et al. (1994). Mathematical Logic. Springer.
[27] Church A. (1932). A set of Postulates for the Foundation of Logic. Annals of Mathematics, second series 33, 346-366.
[28] Andrews P. (2002). An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Springer.
[29] Bergmann E. y Noll H. (1977). Mathematische Logik mit Informatik-Anwendungen. Springer.
[30] Richter M. (1978). Logikkalküle. Teubner.
[31] Gallier J. (1987). Logic for Computer Science. John Wiley.
[32] Schönig U. (1987). Logik für Informatiker, Spektrum Akad. Verlag.
[33] Börger E. (1989). Computability, Complexity, Logic. North-Holland.
[34] Fitting M. (1996). First-Order Logic and Automated Theorem Proving. Spinger.
[35] Nerode A. y Shore R. (1997). Logic for Applications. Springer.
[36] Pásztor K. y Várterész M. (2003). Mathematical Logic: An Application Oriented Approach. Panem Kiadó.
[37] Huth M. y Ryan M. (2004). Logic in Computer Science - Modelling and Reasoning about Systems. Cambridge University Press.
[38] Barr A. y Feigenbaum E. (1981). Handbook of Artificial Intelligence, vols. 3. Heuristic Press.
[39] Bledsoe W. y Loveland D. (1984). Automatic Theorem Proving after 25 years. American Mathematical Society.
[40] Boyer R. y Moore J. (1988). A Computational Logic Handbook. Academic Press.
[41] Gabbay D. et al. (1993). Handbook of Logic in Artificial Intelligence and Logic Programming. Oxford University Press.
[42] Abramsky S. et al. (1995). Handbook of Logic in Computer Science, vols. 1-4. Oxford University Press.
[43] Agostino M. et al. (1999). Handbook of Tableau Methods. Kluwer.
[44] Krantz S. (2002). Handbook of Logic and Proof Techniques for Computer Science. Birkhäuser.
[45] Goguen H. y Goubault J. (2000). Sequent combinators: a Hilbert system for the lambda calculus. Mathematical Structures in Computer Science 10(1), 1-79.
[46] Küsters R. (2001). Non-Standard Inferences in Description Logics. Springer.
[47] Qualls J. y Sherrell L. (2010). Why computational thinking should be integrated into the curriculum. Journal of Computing Sciences in Colleges 25(5), 66-71.
[48] Hoare C.A.R. (2004). Communicating Sequential Processes. McGraw-Hill.
[49] Wing J. (2008). Five deep questions in computing. Communications of the ACM 51(1), 58-60.
[50] Denning P. (2009). The Profession of It: Beyond Computational thinking. Communications of the ACM 52(6), 28.

CAPÍTULO VI
La abstracción como componente crítico de la formación en Ciencias Computacionales[1]

Edgar Serna M.
Instituto Antioqueño de Investigación

Es un hecho que algunos ingenieros de software y científicos computacionales son capaces de producir diseños y programas claros y elegantes, mientras que otros no pueden. Acaso, ¿esto será cuestión de inteligencia? ¿Será posible mejorar en los estudiantes estas aptitudes y habilidades mediante formación y entrenamiento? En este trabajo se explora respuestas a estas preguntas y se argumenta que para los profesionales y estudiantes de Ciencias Computacionales es crucial que posean una buena formación y desarrollen habilidades en abstracción.

[1] Publicado en la Revista Avances en Sistemas e Informática 8(3), 79-83. 2011.

INTRODUCCIÓN

Por más de 30 años el autor ha estado involucrado en la investigación, formación y difusión de las Ciencias Computacionales y de la Ingeniería del Software. Su experiencia en formación abarca cursos como principios de programación, arquitectura de sistemas, arquitectura de software, Ingeniería del Software, Sistemas de Información, algoritmos distribuidos y diseño de software. Cursos que requieren habilidades para analizar, conceptualizar, modelar y resolver problemas. La experiencia en estos años le ha permitido concluir que los estudiantes que sobresalen en Ciencias Computacionales son capaces de: 1) manejar la complejidad de los problemas, para producir modelos y diseños elegantes, y 2) hacerle frente a la complejidad de los algoritmos distribuidos, a la aplicabilidad de diversas notaciones de modelado y a otras cuestiones importantes en esta área.

Por otro lado, existen otros estudiantes que no sobresalen tanto como aquellos, ya que encuentran que los algoritmos distribuidos son muy difíciles, no aprecian la utilidad del modelado, tienen dificultades para identificar lo que es importante en un problema y, por tanto, presentan soluciones complicadas que replican las complejidades mismas del problema. ¿Por qué? ¿Qué es lo que hace que algunos estudiantes puedan lograrlo? ¿Qué les falta a los que no pueden? ¿Es alguna cuestión de inteligencia? Algunos estudios demuestran que la clave está en el desarrollo de capacidades en abstracción, es decir, en la capacidad para realizar y aplicar pensamiento abstracto y poseer habilidades en abstracción.

En este capítulo se explora esta hipótesis y se formula recomendaciones para trabajos futuros. En primer lugar, se discute qué es la abstracción y su papel en las Ciencias Computacionales y otras disciplinas; se utiliza los resultados del desarrollo cognitivo y se analiza los factores que afectan la capacidad de los estudiantes para hacerle frente a la abstracción y para aplicarla. Luego, se discute si la abstracción es o no enseñable y, finalmente, se sugiere los pasos necesarios para poner a prueba las habilidades en abstracción, como medio para validar la hipótesis planteada, revisar la actuales técnicas de enseñanza y, tal vez, mejorar la capacitación de los estudiantes.

1. DEFINICIÓN E IMPORTANCIA DE LA ABSTRACCIÓN

Las diversas definiciones de abstracción [1] se centran en dos aspectos particularmente pertinentes [2]. El primero hace hincapié en el proceso de *eliminar detalles* para simplificar y concentrar la atención, con base en:

- El acto de retirar o remover algo.
- El acto o proceso de no considerar una o más propiedades de un objeto complejo a fin de atender las demás.

El segundo hace hincapié en el proceso de *generalización* para identificar el núcleo común o esencial, con base en:

- El proceso de formulación general de conceptos para abstraer propiedades comunes de las instancias.
- Un concepto general formado por la extracción de características comunes a partir de ejemplos específicos.

La abstracción es ampliamente utilizada en otras disciplinas, como el arte, la escritura y la música. Por ejemplo, es famosa la pintura de Henri Matisse: Naked blue IV, en la que logra representar

con claridad la esencia de su tema, una mujer desnuda, utilizando solo líneas simples o recortes. Su representación elimina todos los detalles, pero transmite mucho. Del mismo modo, Katsushika Hokusai, en su pintura South Wind, Clear Sky, utiliza un equilibrio perfecto de color y composición representando una forma abstracta de la montaña para capturar su esencia. Otro ejemplo es el jazz, en el que los músicos identifican las melodías esenciales o el corazón de una pieza musical en particular, e improvisan a su alrededor, de tal forma que proporcionan sus propios adornos. De acuerdo con los músicos de jazz es fácil hacer complejo a un sonido simple, pero es más difícil hacer simple a un sonido complejo. Esta dificultad es un claro ejemplo del desafío en la aplicación de la abstracción para eliminar detalles superfluos.

Otro ejemplo de la utilidad de la abstracción lo proporcionó Harry Beck [3] en su mapa del metro de Londres de 1928. El mapa era esencialmente una superposición de la red del metro sobre un mapa geográfico convencional de Londres, que mostraba las curvas de las líneas de tren y las del río Támesis y las distancias relativas entre las estaciones. En 1931 Beck produjo la primera representación abstracta y esquemática: simplificó las curvas a solo líneas horizontales, verticales y diagonales, donde las distancias entre las estaciones ya no eran proporcionales a las distancias geográficas. Esta forma de representación simplificada, o abstracción, resultó ser tan adecuada para navegar alrededor del metro de Londres, que desde entonces se ha utilizado para sistemas de transporte en muchos otros países.

El nivel de abstracción utilizado fue cuidadosamente seleccionado a fin de incluir solo los detalles necesarios, pero abandonando los innecesarios: si es demasiado abstracto no proporciona suficiente información para este propósito, y si es demasiado detallado se vuelve confuso e incomprensible. Al igual que cualquier abstracción, puede ser engañosa si se utiliza para otros propósitos: este mapa del metro a veces es mal utilizado por los turistas, ya que lo malinterpretan como a un verdadero mapa geográfico de Londres. Por lo que el nivel, los beneficios y el valor de una abstracción en particular dependen de sus propósitos.

Ahora bien, ¿por qué es importante la abstracción en las Ciencias Computacionales y en la Ingeniería del Software? El software en sí ciertamente es abstracto y el desarrollo de software requiere habilidades de abstracción. Deblin [4] señala que *una vez que te das cuenta de que las Ciencias Computacionales tienen que ver con la construcción, manipulación y razonamiento acerca de abstracciones, se hace evidente que un pre-requisito importante para la buena escritura de programas de computador es la capacidad para manejar abstracciones de manera precisa*.

Wing [5] confirma la importancia de la abstracción en el pensamiento computacional haciendo hincapié en la necesidad de pensar en múltiples niveles de abstracción. Ghezzi et al. [6] identifican a la abstracción como uno de los principios fundamentales de la Ingeniería del Software para dominar la complejidad. Autores como Hazzan [7] también han discutido la abstracción como un pilar básico para las matemáticas y la computación. Además, la eliminación de detalles innecesarios es evidente en la Ingeniería de Requisitos y en el diseño de software.

La elicitación de requisitos consiste en identificar los aspectos críticos del entorno que requiere el sistema, mientras se abandona los irrelevantes. El diseño requiere que se evite la implementación innecesaria de restricciones, por ejemplo, en el diseño de un compilador a menudo se emplea una sintaxis abstracta para centrarse en las características esenciales de la construcción del lenguaje, y se diseña el compilador para producir código intermedio para una máquina abstracta idealizada, con la idea de mantener la flexibilidad y para evitar la innecesaria dependencia de la misma. El aspecto de generalización de la abstracción se puede ver claramente en el desarrollo, en el uso de abstracciones de datos y de clases en la Programación Orientada por Objetos. La

interpretación abstracta para analizar el programa es otro ejemplo de generalización, donde el dominio del programa concreto se asigna a un dominio abstracto para capturar la semántica computacional para analizar el programa.

Las habilidades para la abstracción son esenciales en la construcción de modelos, diseños e implementaciones apropiadas, que son aptas para el propósito particular que nos ocupa. El pensamiento abstracto es fundamental para manipular y razonar sobre abstracciones, ya sean modelos formales para el análisis o programas en un lenguaje de programación. De hecho, la abstracción es fundamental para las matemáticas y para la ingeniería en general, jugando un papel crítico en la interpretación adecuada de los problemas, para luego producir modelos para el análisis y en la producción de soluciones.

2. LAS CAPACIDADES DE LOS ESTUDIANTES DE CIENCIAS COMPUTACIONALES

¿De qué habilidades en abstracción dependen nuestros estudiantes para su desarrollo cognitivo? ¿Podemos mejorar sus capacidades y, en caso afirmativo, cómo? ¿Es posible formar en habilidades del pensamiento abstracto y de la abstracción?

Jean Piaget [8, 9] sentó las bases para una comprensión del desarrollo cognitivo de los niños, desde que son bebés hasta la edad adulta. Con base en estudios de caso derivó cuatro etapas para el desarrollo: senso-motriz, pre-operacional, operacional concreta y operacional formal. Las primeras dos fases van desde la infancia hasta la primera infancia, cerca de los siete años, cuando el niño demuestra, más o menos, su inteligencia mediante actividades motrices, y luego con el lenguaje y la manipulación temprana de símbolos. La tercera es la etapa operacional concreta, entre los siete y doce años, donde demuestra, más o menos, su inteligencia mediante una comprensión de la conservación de la materia, de la causalidad y de una habilidad para clasificar objetos concretos. La cuarta es la etapa operacional formal, desde los doce años hasta la edad adulta, donde los individuos demuestran una habilidad para pensar de forma abstracta, sistemática e hipotética, y utilizan símbolos relacionados con conceptos abstractos. Esta es la etapa crucial en la que el individuo es capaz de pensar abstracta y científicamente.

Aunque existe algunas críticas acerca de la forma como Piaget realizó sus investigaciones y derivó su teoría, existe un apoyo general para sus ideas fundamentales. Además, estudios y evidencias experimentales apoyan la hipótesis de Piaget de que los niños progresan a través de las tres primeras etapas de desarrollo; sin embargo, parece que no todos los adolescentes progresan de la misma manera hasta la etapa operacional formal a medida que maduran. El desarrollo biológico puede ser un pre-requisito, pero pruebas realizadas en poblaciones de adultos indican que solo entre el 30 y el 35% de los adultos alcanzan la etapa operacional formal [10], y que puede ser necesarias condiciones particulares del medio ambiente y de formación para que tanto adolescentes como adultos alcancen esta etapa.

3. LA FORMACIÓN EN ABSTRACCIÓN

A pesar de que el nivel de exigencia para alcanzar la fase operacional formal de Piaget es bajo, puede ser decepcionante, ya que no parece haber alguna esperanza de poder mejorar el rendimiento de los estudiantes mediante la creación de un ambiente formativo adecuado. Por ejemplo, Huitt y Hummel [9] basados en Woolfolk y McCune-Nicolich [11], recomiendan que para adolescentes se debe utilizar técnicas de formación como darles la oportunidad de explorar muchas cuestiones hipotéticas, animarlos a explicar cómo resuelven los problemas y enseñarles conceptos generales, en lugar de solo hechos.

¿Qué pasa con los contenidos curriculares y los planes de estudios? Es recomendable que, en cualquier pregrado en Ciencias Computacionales, se ofrezcan módulos opcionales de cursos diferentes entre los que se incluya un número adecuado de especialización. Ninguno de esos cursos debe ser sobre abstracción, sin embargo, *todos deben depender de o utilizar la abstracción para explicar, modelar, especificar, razonar o resolver problemas*. Esto podría confirmar que la abstracción es un aspecto esencial de las Ciencias Computacionales, pero que debe trabajarse indirectamente, a través de otras temáticas.

Nuestra experiencia es que las matemáticas son un excelente vehículo para formar en pensamiento abstracto. En los primeros años de algunos pregrados, cuando los planes de estudios tienen más contenido en esta área, parece que a los estudiantes les hicieran falta habilidades de abstracción, y que son menos capaces de lidiar con problemas complejos. Devlin [4] confirma esta tesis al subrayar que *el principal beneficio de aprender y utilizar matemáticas no son los contenidos específicos, sino el hecho de que se desarrolla la capacidad para razonar precisa y analíticamente acerca de estructuras abstractas definidas formalmente*. El movimiento en favor de un tratamiento matemático de las Ciencias Computacionales y de la inclusión de tópicos matemáticos en el currículo es muy fuerte.

Sin embargo, en estas áreas no solo es fundamental que los estudiantes sean capaces de manipular formalismos simbólicos y numéricos, también es necesario que tengan habilidades para pasar del mundo real, informal y complicado, a un modelo abstracto simplificado. El currículo en Ciencias Computacionales de la ACM [12] reconoce la importancia de la abstracción mediante la inclusión de aspectos como encapsulamiento, niveles de abstracción, generalización y clases de abstracción; sin embargo, es el modelado y el análisis del software los que reciben mayor atención.

Por otro lado, la modelización y el análisis formal son básicos para la práctica del pensamiento abstracto y la consolidación de la capacidad de los estudiantes para aplicar la abstracción. El modelado es la técnica de ingeniería más importante: los modelos ayudan a comprender y analizar los problemas grandes y complejos. Dado que los modelos son una simplificación de la realidad, con el propósito de promover la comprensión y el razonamiento, los estudiantes deben ejercitar todas sus capacidades de abstracción para construir modelos, que sean idóneos para un propósito.

También deben ser capaces de trazar mapas entre la realidad y la abstracción, a fin de que puedan apreciar las limitaciones de ésta última y de interpretar las implicaciones del análisis del modelo. La motivación del estudiante se puede mejorar mediante la presentación matemática de los formalismos del modelado, pero de forma problematizadora, de tal forma que pueda beneficiarse de la disposición de herramientas de soporte, como las de comprobación de modelos, para el razonamiento y el análisis.

La experiencia personal del autor le ha demostrado que la construcción de modelos y análisis como parte de un curso en concurrencia [13], puede ser muy alentador. Dado un modelo, los estudiantes manifiestan mayor interés por clarificar los aspectos importantes del problema y por razonar acerca de sus propiedades y comportamiento. Sin embargo, a algunos todavía les resulta extremadamente difícil construir los modelos desde el principio. No es suficiente pensar en lo que desean modelar, necesitan pensar acerca de cómo (el propósito) van a utilizar ese modelo. Aunque son capaces de pensar y razonar en abstracto, estos estudiantes parecen carecer de las habilidades necesarias para aplicar la abstracción.

4. PROPUESTA DE SOLUCIÓN

Si la abstracción es una habilidad clave en las Ciencias Computacionales, ¿en qué se debería centrar los procesos de formación para asegurar que sea efectiva y para que los profesionales tengan adecuados conocimientos para su aplicación y perfeccionamiento? Hasta el momento hemos presentado situaciones principalmente anecdóticas, con alguna evidencia soportada en la literatura, pero ¿cómo sustentar esto con una base más científica para mejorar nuestra comprensión de la situación? Como en todas las actividades científicas e ingenieriles, antes de que podamos controlar o efectuar, primero debemos medir. El objetivo es reunir los siguientes datos:

1. Mientras estén en la universidad, medir anualmente las capacidades en abstracción de los estudiantes. Esto se podría utilizar para comprobar si su habilidad se correlaciona con sus calificaciones y para relacionarlo con los resultados de sus compañeros de semestre. Suponiendo que nuestras técnicas de clasificación convencional (trabajos de cursos, trabajos de laboratorio y evaluaciones) son indicadores de la capacidad de un estudiante, esto ayudaría a ganar confianza en que la abstracción es un indicador clave de capacidad. Un segundo objetivo de estas pruebas es que proporcionarían un medio alternativo para revisar las capacidades del estudiante. Por último, también podría ayudar a evaluar la eficacia de nuestro modelo de enseñanza y para asegurar que las capacidades de todos los estudiantes mejoren a medida que avanzan en su carrera.

2. Medir las capacidades en abstracción de los estudiantes al momento que tramitan su inscripción para estudiar Ciencias Computacionales. Actualmente, el ingreso a las universidades se sustenta casi que exclusivamente en las calificaciones escolares previas. La capacidad en abstracción podría, potencialmente, servir para detectar a aquellos estudiantes que tienen menor probabilidad de lograr un adecuado desempeño en sus estudios, para seleccionar a los que, además de ser académicamente aptos, también tienen una aptitud real para las Ciencias Computacionales y para la Ingeniería del Software.

La realización de estos experimentos y recolección de estos datos depende de la disponibilidad de *buenas pruebas en abstracción*, para medir el pensamiento y las capacidades de los estudiantes acerca de la misma. Desafortunadamente, no hemos podido detectar la existencia de pruebas apropiadas. Las pruebas para la etapa operacional formal se enfocan principalmente en el razonamiento lógico, pero no son apropiadas para probar capacidades en abstracción ni para distinguir entre las capacidades de los estudiantes de un nivel universitario.

Dubinsky y Hazzan [14] recomiendan diseñar un conjunto específico de preguntas de prueba, incluyendo suficientes y diferentes tipos de tareas y descripciones, soportadas en la recolección de datos cuantitativos y cualitativos, e incluyendo preguntas y entrevistas no limitadas de antemano. Estas pruebas deben examinar las diferentes formas y niveles de abstracción y los diferentes propósitos para esas abstracciones. Este debe ser nuestro siguiente paso. Solo entonces podremos tener datos definitivos en cuanto a la criticidad de la abstracción en las Ciencias Computacionales y de nuestra habilidad para formar en ella. Por ejemplo, deberíamos ser capaces de confirmar o refutar que un curso en particular, como el modelado y el análisis formal, es realmente un medio efectivo para enseñar abstracción.

5. CONCLUSIONES

Al igual que muchos, creemos que la abstracción es una habilidad clave para las Ciencias Computacionales y que es esencial en la Ingeniería de Requisitos para elicitar los aspectos críticos

del entorno que requiere el sistema. En el diseño necesitamos articular la arquitectura del software y la funcionalidad de los componentes, de tal forma que satisfagan los requisitos funcionales y no-funcionales, mientras evitamos innecesarias restricciones de implementación. Incluso, en la fase de implementación, utilizamos la abstracción de datos y de clases con el fin de generalizar las soluciones.

En este trabajo hemos propuesto que la razón por la que algunos ingenieros de software y científicos computacionales son capaces de producir diseños y programas claros y elegantes, mientras que otros no lo son tanto, se puede atribuir al desarrollo de sus capacidades abstractivas. Argumentamos que una buena comprensión del concepto de abstracción, y su importancia en la formación en Ingeniería del Software, es crucial para el futuro de las Ciencias Computacionales. Lo primero que se necesita es un conjunto de pruebas en abstracción para revisar el progreso del estudiante, que permita comprobar nuestras didácticas y potencializarlas como una ayuda para seleccionar estudiantes en los procesos de admisión.

REFERENCIAS

[1] Webster. (2002). Third New International Dictionary of the English Language. Merriam-Webster.

[2] Frorer P. et al. (1997). Revealing the faces of abstraction. International Journal of Computers for Mathematical Learning 2, 217-228.

[3] BBC News. (2006). Top three “iconic” designs named. Recuperado: http://news.bbc.co.uk/2/hi/uk_ news/england/london /4769060.stm

[4] Devlin K. (2003). Why universities require computer science students to take math. Communications of ACM 46(9), 37-39.

[5] Wing J. (2006). Computational thinking. Communications of ACM 49(3), 33-35.

[6] Ghezzi C. et al. (2003). Fundamentals of Software Engineering. Pearson.

[7] Hazzan O. (1999). Reducing abstraction level when learning abstract algebra concepts. Educational Studies in Mathematics 40, 71-90.

[8] Piaget J. y Inhelder B. (1972). The Psychology of the Child. Basic Books.

[9] Huitt W. y Hummel J. (2003). Piaget’s theory of cognitive development: Educational Psychology Interactive. Valdosta State University Press.

[10] Kuhn D. et al. (1977). The development of formal operations in logical and moral judgment. Genetic Psychology Monographs 95, 97-188.

[11] Woolfolk E. y McCune L. (1984). Educational Psychology for Teachers. Prentice-Hall.

[12] ACM, AIS, IEEE-CS. (2005). The ACM/IEEE Computing Curricula. ACM & IEEE Press.

[13] Magee J. y Kramer J. (2006). Concurrency - State Models and Java Programs. John Wiley & Sons.

[14] Dubinsky Y. y Hazzan J. (2004). Shaping of a teaching framework for software development methods at higher education. En The International Workshop on Learning and Assessment in Science, Engineering & Management in Higher Education. Haifa, Israel.

CAPÍTULO VII
Enfoque de la lógica y la abstracción en la formación en ingeniería[1]

Edgar Serna M.[1]
Luis Fernando Zapata A.[2]
[1]Instituto Antioqueño de Investigación
[2]Corporación Universitaria Remington

Este capítulo trata acerca de la lógica y la abstracción vistas como una relación necesaria en los procesos formativos de los ingenieros. Se describe la importancia y la necesidad de formar en esta área del conocimiento, y de la relación entre el ejercicio profesional de los ingenieros y el desarrollo y/o potencialización de su capacidad lógico-interpretativa y abstractiva para la resolución de problemas. Es un trabajo de investigación-revisión-reflexión acerca de la importancia de estos dos componentes, vistos estructuralmente desde el currículo, porque si el objetivo de formar es ofrecerle a la sociedad profesionales que generen confianza, lo recomendable sería modificar los procesos formativos a los cuales están expuestos. La Sociedad de este siglo necesita profesionales confiables desde lo ético y lo humano, pero fundamentalmente solucionadores de problemas, y es ahí donde cobran importancias la lógica y la abstracción.

[1] Publicado en la Revista Internacional de Educación y Aprendizaje 2(1), 35-47. 2014.

INTRODUCCIÓN

En el mundo del siglo XXI la ingeniería se concibe como una disciplina cada vez más dominada por las técnicas de modelización, una práctica que requiere procesos como comprender el problema, abstraer, modelar, construir y evaluar los diseños antes de la fabricación física de una solución. Además, la mayoría de los sectores productivos les exige a los ingenieros habilidades especiales para su ejercicio profesional, como pensamiento lógico-sistémico, resolución de problemas y capacidad de abstracción.

Si bien las personas se sienten atraídas por la ingeniería porque les gusta aplicar la ciencia y usar sus habilidades para resolver problemas, la formación que se imparte en la mayoría de programas no orienta a los estudiantes en pro de alcanzar ese objetivo, por lo que la recomendación es estructurar e implementar currículos que orienten al desarrollo y/o potencialización de sus capacidades en lógica y abstracción, con la meta de formar y capacitar profesionales creativos en ingeniería y con las habilidades, destrezas, conocimiento y capacidades necesarias para resolver los problemas de la sociedad de este siglo. David Parnas, un pionero de la Ingeniería del Software, dijo alguna vez que para los ingenieros es esencial una sólida formación y comprensión de la lógica y la abstracción, porque junto a la ingeniería son áreas que no se pueden interpretar libremente.

La ingeniería trata acerca de los procesos necesarios para construir cosas, generalmente con un propósito preconcebido, y quien la practica debe utilizar su ingenio para lograrlo. La abstracción es un proceso mental para eliminar detalles con el objetivo de centrarse en lo realmente importante del problema, para generar un modelo abstracto de la solución. De otro lado, la lógica trata acerca de la esfera de una verdad formal a priori, abarca a las matemáticas y es crucial para la ingeniería, porque es la base sobre la que se soporta la construcción y explotación de los modelos abstractos o matemáticos.

La capacidad para resolver problemas es un componente importante en el ejercicio profesional de los ingenieros, e inclusive puede ser el núcleo de su ejercicio. En este siglo, y como nunca antes en la historia, esta capacidad está dominando cada vez más el contenido intelectual para desarrollar esta capacidad, y los principios de la lógica y de la abstracción cobran especial importancia para potencializarla. Las soluciones propuestas a los problemas actuales serán más eficaces si se sustentan en procedimientos y modelos construidos con sólidos fundamentos lógicos, pero en los procesos formativos el desarrollo y aplicación de la lógica apenas si se prevé, y la responsabilidad se delega a las matemáticas como un único núcleo alrededor del cual gira los procesos ingenieriles.

La logización de estos procesos debe ser una etapa natural, porque esto les permitirá a los ingenieros aprovechar de mejor manera sus habilidades y destrezas para resolver los cada vez más complicados problemas de la sociedad. Si bien la lógica y la abstracción son cuestiones necesarias para que un estudiante comprenda el mundo y potencialice sus habilidades, la mayoría de contenidos curriculares apenas si las mencionan, no se estructuran adecuadamente uno con otro, ni se proponen como fundamento para desarrollar capacidades en los estudiantes, aunque esta posibilidad es una parte importante de la motivación para que se decidan a iniciar programas en esta área de formación.

En este capítulo se describe la importancia y la necesidad de formar adecuadamente a los ingenieros en lógica y abstracción, con el objetivo de capacitarlos para comprender, analizar y modelar los problemas del siglo XXI, y para que les presenten soluciones eficientes y eficaces.

1. ALGUNAS CUESTIONES CLAVE

Resolver problemas es una de las habilidades más importante en la que se debe formar a los estudiantes en cualquier parte del mundo. En los contextos laborales, a los profesionales se les paga para resolver problemas, y la vida cotidiana gira constantemente alrededor de la resolución de los mismos [1]. Cada día nos enfrentamos a problemas, grandes y pequeños, simples y complejos, claros y confusos, pero paradójicamente en las instituciones de educación los procesos parecen ignorar en gran medida la necesidad de desarrollar y/o potencializar la capacidad lógico-interpretativa y abstractiva en los estudiantes.

En la revisión a la literatura que se hizo prácticamente no se encontraron referencias a cómo desarrollar esa capacidad, y muy poco acerca del diseño instruccional orientado a la resolución de problemas. En las primeras ediciones de su libro, Gagné [2] se refería a la resolución de problemas, pero al parecer en las ediciones posteriores se dio *por vencido,* prefirió enfrentarse con las reglas de orden superior y se adentró en otros terrenos. ¿Por qué? Si los problemas son pandémicos y solucionarlos es esencial para la actividad cotidiana y profesional ¿por qué no se hace un mayor esfuerzo para formar a los estudiantes para resolverlos adecuadamente?

Por mucho tiempo algunos modelos de procesamiento de la información de resolución de problemas han tratado de explicar y aplicar una metodología para formar en esta cuestión. Ejemplos de este trabajo son: General Problem Solver [3], un modelo que especifica dos tipos de procesos de pensamiento asociados con la capacidad para resolver problemas: 1) comprender el contexto, y 2) aplicar procesos de búsqueda; otro es el IDEAL [4], que describe la resolución de problemas como un proceso uniforme en etapas: 1) identificar problemas potenciales, 2) definirlos y representarlos, 3) explorar posibles estrategias de solución, 4) actuar de acuerdo con esas estrategias, y 5) mirar hacia atrás y evaluar los efectos de esas actividades. Aunque este modelo supone que esos procesos se aplican de manera diferente para cada problema no presenta sugerencias explícitas acerca de *qué hacer* para *saber cómo* hacerlo.

Gick [5] sintetiza estos y otros modelos y los simplifica en: 1) construir una representación del problema, 2) buscar soluciones, y 3) aplicar y hacer seguimiento a las soluciones. Aunque estas propuestas son útiles descriptivamente, tienden a tratar de la misma forma a todos los problemas, en un esfuerzo por articular un procedimiento generalizado. La culminación de los conceptos de procesamiento de la información [6] fue otro intento, aunque sin éxito, por articular una teoría uniforme para solucionar problemas.

La suposición subyacente es que la habilidad para resolver problemas es un tipo especial de formación, especialmente para ingenieros, que resulta de desarrollar la capacidad lógico-interpretativa y abstractiva de los estudiantes. De otro lado, existe diferentes formas para resolver problemas, y cada una requiere distintos tipos de habilidades y destrezas en lógica y abstracción. Estas habilidades son las que se adquiere al desarrollar esa capacidad y las que capacitan a un estudiante para adaptarse a una situación problemática, interpretarla, comprenderla, modelarla, y luego presentarle una solución eficiente y eficaz.

Por otro lado, al hablar de los *ingredientes* necesarios para que los ingenieros resuelvan problemas con éxito, se considera útil hacer una distinción entre los principios científicos y las técnicas de resolución de problemas. Los primeros se orientan a las leyes, como la de conservación de la masa, las de los gases, la de Ohm, la de Hooke, las de la termodinámica, y así sucesivamente; las segundas incluyen el uso de modelos matemáticos algebraicos y gráficos, la lógica simbólica, la capacidad de abstracción, los diagramas de flujo, el juicio, las técnicas de

solución de errores y los programas informáticos, entre otros muchos. Por supuesto, ambos principios se sustentan en bases sólidas de lógica y abstracción.

Además, generalmente se reconoce que los ingenieros deben utilizar determinadas herramientas matemáticas y lógicas para realizar su función principal: *resolver problemas* y que, al igual que con los principios científicos, las utilizan para correlacionar los diferentes componentes del contexto de la situación problemática. En estos ambientes los principios y herramientas se deben comprender desde dos puntos de vista para seleccionar los datos correlacionales necesarios: 1) con referencia a su uso en la solución de problemas específicos, y 2) desde examinar el fondo de los conocimientos disponibles sobre el contexto. Algunas de esas herramientas y métodos son:

- Las matemáticas
- La lógica
- La abstracción
- El modelado
- Los modelos matemáticos
- Los procedimientos gráficos
- Los conceptos de gráficos de flujo
- La iteración
- Las técnicas de solución mediante prueba y error

Para aplicar eficiente y eficazmente estas herramientas y métodos, los ingenieros necesitan desarrollar su capacidad lógico-interpretativa y abstractiva, que les permita comprender los problemas para presentarles una solución acorde. Esta característica representa una marcada diferencia entre ellos y otros profesionales, porque son quienes más la aplican, aunque diversas investigaciones y estudios han demostrado que el ejercicio profesional en cualquier área del conocimiento necesitará en algún momento de ella. Ese proceso requiere la comunión constante entre tres áreas clave: Ingeniería, lógica y abstracción, que deben hacer parte de sus procesos formativos para formar y capacitar a los profesionales que la sociedad necesita.

2. LÓGICA Y ABSTRACCIÓN: ESTADO DEL ARTE

A mediados del siglo XX algunos investigadores evaluaron el nivel de comprensión, y el alcance y las limitaciones de dominio que las personas alcanzan del desarrollo de la lógica [7-15]. Sin embargo, esos estudios no reportaron en qué medida las personas comprenden el significado lógico de la temática investigada, no definieron explícitamente la situación experimental aplicada, y no proyectaron el uso de los resultados para proponer cambios en los procesos formativos. Del mismo modo, y en la medida que las edades de la población se incrementan, también se requiere información más detallada acerca de sus hábitos lingüísticos y de la comprensión lógica que aplican.

Mediante dos experimentos relacionados, Suppes y Feldman [16] determinaron la forma en que los niños de edad preescolar comprenden el significado de los conectores lógicos y de la lógica misma, con lo que contribuyeron a la acumulación de información sistemática en esta área. McCarthy y Hayes [17] propusieron que para que un programa de computador fuera capaz de actuar *inteligentemente*, debía tener una representación general del mundo en términos de las entradas que debe interpretar. Además, que para diseñarlo se requería saber *qué* es y *cómo* se obtiene el conocimiento, un proceso que requiere lógica y abstracción. Sloam [18] respondió a este trabajo argumentando que los problemas filosóficos acerca del uso de la *intuición* en el razonamiento, relacionados a través de un concepto de representación analógica para problemas

como la simulación de la percepción, la resolución de problemas y la generación de conjuntos útiles, se debían estudiar considerando una manera específica de actuar. Concluyó que los requisitos propuestos por McCarthy y Hayes para tomar decisiones inteligentes eran demasiado estrechos, y en su lugar propuso requisitos más generales. Pero en su trabajo no menciona ni aplica la conceptualización lógico-abstractiva como base para lograrlo.

La capacidad para pensar de forma abstracta es una habilidad necesaria para el desarrollo profesional, pero es común encontrar diferencias en la forma como la aplican los estudiantes, porque algunos la desarrollan con mayor facilidad que otros, e incluso algunos no llegan a desarrollarla [19]. Estos estudios demostraron que existe vínculos entre la capacidad abstractiva y el éxito en la resolución de problemas, por lo que desarrollarla debería ser un objetivo de los procesos formativos. La capacidad para pensar de esta forma fue identificada por Piaget [20] como una de las etapas del desarrollo cognitivo de los niños, y describió la cuarta y última etapa de este proceso como de *operación formal,* observando que solo alrededor del 35% de ellos la alcanza.

En los años 80 se realizaron varias investigaciones aplicando modelos computacionales alrededor de las temáticas de la resolución de problemas, del modelado cognitivo y de la memoria a largo plazo. Entre ellas se encuentran las de Schank [21], Kolodner [22] y Ross [23], cuyos resultados les sirvieron a Kolodner et al. [24] para explorar las formas en que el razonamiento basado en casos puede ayudar en esa resolución. De acuerdo con su modelo la transferencia de conocimiento entre casos es en gran medida guiada por el mismo proceso de la resolución, además, demostró las interacciones entre los procesos de resolución y la memoria por experiencia. Su programa de computador MEDIATOR ilustra el razonamiento basado en casos para interpretar y resolver disputas de sentido común. El resultado de este trabajo fue un modelo de razonamiento basado en casos que integra la solución de problemas, la comprensión y la memoria, pero que no hace referencia a cómo integrar el proceso mental lógico-abstractivo que las personan aplican para resolver esos problemas.

A finales de siglo se investigó activamente acerca del uso del razonamiento abductivo, y se presentaron trabajos como el de O'Rorke [25], cuyo tema central fue el aprendizaje y el descubrimiento, el de Dasigi y Reggia [26] acerca del procesamiento del lenguaje natural, y el de Peng y Reggia [27] relacionado con el diagnóstico de errores. Estos estudios les sirvieron a Kumar y Venkataram [28] para proponer un modelo con el objetivo se resolver problemas de diagnóstico, fundamentado en el mecanismo de inferencia abductiva, que se aplicó para añadir algunas características nuevas al modelo general existente de resolución de problemas de diagnóstico. Estos investigadores fueron los primeros en combinar las matemáticas con los algoritmos computacionales, y sus resultados demostraron efectividad en la resolución de los problemas de diagnóstico, aunque no se acercaron a la lógica y a la abstracción como herramientas que también trabajan desde la abducción.

Por su parte, Gottinger y Weimann [29] exploraron las diversas técnicas de inferencia para un sistema de apoyo a las decisiones inteligentes basadas en diagramas de influencia, y concluyeron que el razonamiento acerca de la acción requiere varios niveles de representación e inferencia, que dependen del nivel de incertidumbre y de la complejidad y de la novedad de la situación de decisión. Para hacerlo se basaron en la caracterización unificada de procedimientos de inferencia de conocimientos de la lógica probabilística, en el razonamiento teórico de decisiones, en el trabajo de Jarke y Radermacher [30] acerca de la lógica de la solución de problemas mediante técnicas para el análisis probabilístico, la toma de decisiones bajo incertidumbre y la investigación de operaciones, y en el de Gottinger y Weimann [31] acerca de técnicas basadas en Inteligencia

Artificial para sistemas de soporte a las decisiones inteligentes. Aunque aplicaron la lógica desde la visión de las decisiones inteligentes, no cubrieron aspectos como la interpretación, el modelamiento de problemas o la capacidad abstractiva de quien intenta tomar esas decisiones. Lo interesante de esta investigación es la aplicación que se hace de los métodos formales.

Morris y Sloutsky [32] intentaron averiguar si el razonamiento abstracto se desarrolla naturalmente, y cómo contribuye a esto los procesos formativos. Su investigación se centró específicamente en los efectos que una formación prolongada tiene en el desarrollo del razonamiento abstracto-deductivo, y más concretamente en el desarrollo de la comprensión de la necesidad lógica. Plantearon la hipótesis de que el énfasis en formar en el meta-nivel de deducción en un dominio de conocimiento, puede mejorar el desarrollo del razonamiento deductivo, tanto *dentro* como *a través* de ese dominio.

Este trabajo consistió de dos estudios en los que se examina el desarrollo de la comprensión de la necesidad lógica en el razonamiento algebraico-deductivo y verbal. Los resultados apoyaron la hipótesis, lo que indica que una formación prolongada, con énfasis en el meta-nivel de la deducción algebraica, contribuye al desarrollo de la comprensión de la necesidad lógica, tanto en el razonamiento algebraico-deductivo como en el verbal. Los resultados también sugirieron que muchos adolescentes, aunque estén expuestos a los mismos procesos de formación, no desarrollan naturalmente una comprensión de esa necesidad lógica.

Pietarinen [33] intentó responder a la cuestión de *qué tienen en común la lógica epistémica y la ciencia cognitiva*, y concluyó que existen tres posibilidades: 1) nuevas versiones cuantificadas de multi-agentes lógicos epistémicos capturan las locuciones de identificación de los objetos involucrados, dando lugar a aplicaciones de la representación de conocimiento en sistemas multi-agente y procesamiento paralelo; 2) el marco de la semántica de la teoría de juegos para las lógicas consiguientes tiene mayor credibilidad cognitiva, como una verdadera semántica para las nociones epistémicas; y 3) algunos hallazgos en neurociencia cognitiva, relacionados con las nociones de conocimiento y transformación explícita vs procesamiento implícito, contribuyen a los estudios de lógica. Estas conexiones las explora el autor desde las perspectivas lógica y cognitiva, y los resultados definieron nuevas extensiones de la lógica epistémica, incrementaron la comprensión formal del procesamiento de la información inconsciente y consciente en el cerebro, y con ellos se logra que el formalismo susceptible de representación del conocimiento se configure en multi-agentes.

Huitt y Hummel [34] afirman que la mayoría de adultos necesita un *entorno especial* para alcanzar la cuarta etapa de desarrollo de Piaget, pero no es claro si lo logran gracias a una habilidad innata con la que nacen, o si se desarrolla a través de los procesos formativos a los que se exponen. Piaget describe esta etapa como la *del uso lógico de símbolos relacionados con conceptos abstractos*, descripción que también se podría utilizar para referir la capacidad de interpretar problemas y de producir modelos, una característica importante para el desarrollo profesional de los ingenieros.

Pietarinen [35] argumenta que los hallazgos empíricos acerca de las raras disfunciones neuronales son contribuciones de las investigaciones en lógica, y que la fase temprana de la ciencia cognitiva comparte raíces con la fenomenología. En consecuencia, identifica líneas en ese período inicial que se originan en la lógica, la IA y las Ciencias Computacionales. Otra conclusión a la que llegó es que desde estas fases también se reconoce la importancia de la división entre aspectos implícitos y explícitos del conocimiento en la cognición comprensiva. Aunque el aporte de este trabajo es importante para reconocer las raíces de la *discapacidad lógica* en algunas

personas, los resultados no se aplican para indicar el mismo proceso en personas sin esa discapacidad.

En la misma línea, Egorov [36] parte de la hipótesis de que la capacidad de pensar lógicamente está determinada por los genes, a los que llama informalmente *genes lógicos* y que hipotéticamente codifican la información de las proteínas. Se pregunta si en los seres humanos realmente existe genes para la lógica, y se responde que lo más probable es que sí, debido a que éstos contribuyen en gran medida al control de la cognición, tal como lo investigaron McLearn et al. [37], Winterer y Goldman [38], Oldham et al. [39], Popesco et al [40] y Reuter et al [41]. Egorov pretende encontrar el origen de la capacidad lógico-interpretativa de las personas, pero, aunque su aporte es importante, no aplica el mismo procedimiento para la capacidad abstractiva, lo que no permite hacer una inferencia relacional del desarrollo de ambas capacidades.

A diferencia de otros animales los seres humanos están equipados con un poderoso cerebro que los dota de conciencia y reflexión, sin embargo, una creciente tendencia en sicología cuestiona los beneficios de esa conciencia [42]. Por otro lado, DeWall et al. [43] aplican cuatro estudios cuyos resultados sugieren que la conciencia, como sistema de procesamiento reflexivo, es importante para el razonamiento lógico, de lo que se puede concluir que el sistema de procesamiento reflexivo *ayuda* a ese razonamiento. Ellos se basaron en esto para presentar su hipótesis de que el razonamiento lógico depende en gran medida del procesamiento consciente, y propusieron que la forma de probar esta teoría sería logrando que las manipulaciones afecten solo a uno u otro de los dos sistemas de procesamiento, y que dejen al otro intacto.

Por otro lado, muchos estudiantes carecen de las habilidades para buscar lógicamente la información que requieren y, aunque existe abundancia de ella a su disposición, no son capaces de leerla, analizarla ni evaluarla críticamente [44]. Para hacerles frente a estos problemas, Bouhnik y Giat [45] desarrollaron un curso universitario con el objetivo capacitar a los estudiantes para aplicar herramientas lógicas. El curso se sirvió a dos grupos de estudiantes diferentes, uno orientado por las áreas sociales y otro por las ciencias exactas. El objetivo fue estudiar y comprender los sistemas lógicos basados en el concepto, y los resultados demostraron que las habilidades en el razonamiento lógico y crítico de los estudiantes en ambos grupos mejoraron con el tiempo, tanto objetiva como subjetivamente. Este trabajo hace varias contribuciones a los campos de la formación en TI, en lógica aplicada y en pensamiento crítico.

De acuerdo con investigadores como Simon [46], Harel y Sowder [47] y Lithner [48], de las disciplinas relacionadas con la formación en matemáticas, como la filosofía, la psicología y la matemática misma, se deriva razonamientos como el inductivo, el deductivo, el abductivo, el plausible y el de transformación. Cañadas et al [49] consideraron la diferenciación general entre los razonamientos inductivo y deductivo, desde la tradición filosófica y desde las diferentes disciplinas y contextos en que esa distinción persiste, y se esforzaron por centrar su investigación en el proceso del razonamiento inductivo, mientras que Ibañez [50], Marrades y Gutiérrez [51] y Stenning y Monaghan [52] resaltan las dificultades prácticas de realizar esa separación. Concluyeron que los estudiantes aplican acciones lógicas con mayor frecuencia en problemas cuyos casos particulares se expresan de forma numérica, y que son capaces de identificar la aplicabilidad de ciertos pasos del razonamiento inductivo que habían utilizado antes en el aula.

En este orden de ideas, Halpern y Pucella [53] examinaron cuatro enfoques para abordar el problema de la omnisciencia lógica y su aplicabilidad potencial: 1) el sintáctico [54-56], 2) el de la conciencia [57], 3) el del conocimiento algorítmico [58], y 4) el de los mundos imposibles [59]. Aunque algunos investigadores aceptan que estos enfoques poseen el mismo nivel de

expresividad y pueden capturar todos los estados epistémicos, otros demuestran lo contrario. El objetivo de la investigación de Halpern y Pucella [53] fue hacerle frente a la omnisciencia lógica, es decir, a cómo elegir un enfoque y construir un modelo apropiado. Teniendo en cuenta la pragmática de esta área y con base en el principio de la falta de omnisciencia lógica en esta situación, identificaron algunos principios que rigen el proceso de cómo elegir un enfoque para una situación modelo, y concluyeron que el enfoque de mundos imposibles es *especialmente adecuado* para representar un punto de vista subjetivo del mundo.

También surgió el interés por investigar en la recopilación de pruebas acerca de los vínculos entre el pensamiento abstracto y el desarrollo de la capacidad abstractiva, y mientras algunas investigaciones concluyen que la lógica y la abstracción son *habilidades clave* para la formación en Ciencias Computacionales e Ingeniería [60], otras tratan de encontrar un vínculo en el éxito de las habilidades de abstracción en los cursos de lógica computacional [61, 62], con logros diversos.

3. LA CAPACIDAD LÓGICO-INTERPRETATIVA Y ABSTRACTIVA

De acuerdo con Andrews [63], ser lógico presupone: 1) tener sensibilidad para el lenguaje y habilidad para utilizarlo efectivamente, porque la lógica y el lenguaje son inseparables, 2) tener gran respeto por el escenario mundial, porque la lógica trata de la realidad, y 3) tener una conciencia viva de cómo se relacionan los hechos (las ideas) con los objetos en el mundo, porque la lógica trata de la verdad. Desarrollar efectivamente estas habilidades, actitudes, puntos de vista y modalidades prácticas le permite a cualquier persona preparar su mente para trabajar con éxito la lógica, pero para lograr eficientemente debe:

- Ser un excelente observador
- Estar atento
- Obtener los hechos directamente
- Comprender las ideas y sus objetos
- Estar consciente de los orígenes de las ideas
- Hacer coincidir las ideas con los hechos
- Hacer coincidir las palabras con las ideas
- Realizar análisis profundos
- Concatenar situaciones para obtener conclusiones
- Comunicarse efectiva y eficientemente
- Evitar el lenguaje vago y ambiguo
- Evitar el lenguaje evasivo
- Tener concentración
- Ser realista
- Buscar la *verdad*
- Habilidades comunicativas en diversas formas
- Ser buen oyente
- Ser buen lector
- Gustarle escribir

La Lógica se refiere a la formalización de las leyes del pensamiento y se centra en la formulación de teorías normativas que establecen *cómo* se debe pensar *correctamente*; se relaciona no solo con el pensamiento abstracto, sino también con el que se representa en forma de oraciones y con el que manipula frases para generar nuevo pensamiento. Si la lógica es una formalización de este tipo, entonces el mejor lugar para encontrarla sería en el cerebro [64], pero limitar la observación a su estructura y actividad sería como analizar el hardware cuando el objetivo es el software, o

como tratar de comprender las interacciones humanas estudiando el movimiento de las partículas atómicas, por lo que se recomienda utilizar el sentido común y tener como base la introspección, aunque sea poco fiable [4]. El optimismo a ultranza puede llevar a que se vea lo que se quiere ver, en lugar de ver lo que realmente está allí, y el modelamiento y la simulación son herramientas a las que se puede recurrir para tener éxito.

De acuerdo con esto: ¿cuáles serían las capacidades lógico-abstractas de las que dependen las personas para su desarrollo cognitivo? ¿Cómo se puede mejorar esas capacidades? ¿Sería posible *enseñar* habilidades de pensamiento lógico-abstracto? Con base en algunos estudios de caso, Piaget e Inhelder [65] y Huitt y Hummel [66] sentaron las bases para una mejor comprensión del desarrollo cognitivo. En su trabajo derivaron cuatro etapas para este proceso: 1) senso-motriz, 2) pre-operacional, 3) operacional concreto, y 4) operacional formal.

En las dos primeras la inteligencia se demuestra primero mediante actividades motrices y, posteriormente, con el lenguaje y la manipulación temprana de símbolos; en la tercera se demuestra mediante la comprensión de la conservación de la materia y de la causalidad, y de una habilidad para clasificar objetos concretos; y en la cuarta se demuestra una habilidad para pensar de forma abstracta, sistemática e hipotética, se utiliza símbolos relacionados con conceptos abstractos, y es una etapa crucial en la que el individuo es capaz de pensar abstracta y científicamente.

Aunque existen estudios y evidencias experimentales que apoyan la hipótesis del progreso a través de las tres primeras etapas, parece que no todas las personas progresan hasta la operacional formal a medida que maduran. El desarrollo biológico puede ser un pre-requisito, pero pruebas realizadas en poblaciones de adultos indican que solo entre el 30% y el 35% alcanza la etapa operacional formal [67]. Además, para que los adolescentes y los adultos la alcancen puede ser necesarias condiciones particulares desde el medio ambiente y procesos formativos.

Hall [68] menciona que la lógica les permite a los humanos examinar las ideas, los conceptos y los procesos mentales, porque se encuentra en todas las esferas de la vida ordinaria. En estas esferas la capacidad lógica y abstracta se expresa utilizando relaciones lógicas en lenguaje natural, un principio necesario para simplificar y comprender la cotidianidad. Wason y Johnson [69] sostienen que la comprensión de reglas y regulaciones se ha convertido en un problema que afecta la vida de las personas, es decir, se refieren a la forma complicada como se presenta las relaciones lógico-abstractas. Para solucionar este inconveniente algunos investigadores sugieren utilizar un *árbol lógico*, en el que sea posible reflejar con claridad todas esas relaciones. La idea es lograr que esa estructura sea lo suficientemente sencilla y clara, luego de eliminar las conexiones complejas entre las cláusulas, porque son las que impiden la comprensión de las reglas lógico-interpretativas y abstractivas involucradas.

Estos conceptos y principios constituyen la base para desarrollar la capacidad lógico-investigativa y abstractiva en los estudiantes en ingeniería, y aunque esta área ha sido poco investigada existe una necesidad generalizada de re-orientar los procesos formativos para alcanzar su desarrollo. Como se ha podido identificar en el análisis al estado del arte, existen tres posibles hipótesis para lograr el desarrollo de esta capacidad: 1) es una cuestión genética, 2) es el resultado de procesos formativos focalizados, y 3) es una combinación de las dos anteriores. Aunque no se encontró la validación de ninguna de ellas, es posible concluir que el desarrollo de la capacidad-lógico interpretativa y abstractiva es un componente básico para el ejercicio profesional, especialmente de los ingenieros.

4. DESARROLLO DE LA CAPACIDAD LÓGICO-INTERPRETATIVA Y ABSTRACTIVA

Idealmente, la práctica de la ingeniería se puede describir como la solución óptima de problemas físico-prácticos mediante un análisis lógico-abstracto y sistemático, y/o la integración de hechos científicos. Sin embargo, el número, la complejidad y la indeterminación de los hechos que se debe manejar en las soluciones es lo suficientemente complejo como para que se agregue invariantemente el *juicio* y la *razón*. El juicio es el objeto de la intuición personal y ha dado lugar a la interpretación del concepto de la ingeniería como un arte antes que como una ciencia puramente lógica. El juicio es sin duda un componente conocido de la práctica de esta profesión, debido a que se debe considerar múltiples factores que a menudo son intangibles. Pero es quizás menos reconocido como un factor significativo en el nivel más bajo de los cálculos ingenieriles, donde solo se presume que están presentes los hechos y la lógica, porque los propios hechos no están completamente claros y normalmente se basan en una serie de hipótesis y supuestos, que a su vez requieren del juicio y la razón.

Como seres humanos, el juicio de los ingenieros está sujeto a influencias emocionales, y aunque gran parte de los procesos formativos se niegue a aceptarlo, la capacidad lógico-interpretativa y abstractiva puede ser una contribución tan importante a las conclusiones y las decisiones resultantes de un estudio de ingeniería, como los hechos y procesos del pensamiento lógico en sí, y esa falta de aceptación de la importancia de la lógica y el pensamiento abstracto impregna todos los resultados de los proceso ingenieriles. Causa de ello es que en el aula rara vez se discute y analiza la influencia de los procedimientos lógicos en los procesos de razonamiento, y en su lugar se acrecientan y, en cierta manera, potencializan las presunciones en las que se basa ese razonamiento.

Este proceso lo describen algunos investigadores como un *condimento de las presunciones injustificadas para conclusiones inevitables*. En otras palabras, cuando un individuo piensa lógicamente, pero influenciado emocionalmente, tiende a seleccionar aquellas suposiciones que lógicamente lo llevan a la conclusión que desea. Esto se podría evitar sin en los procesos formativos de los ingenieros se le da la importancia necesaria a la lógica y a la abstracción, y si se estructuran currículos orientados a desarrollar y/o potencializar la capacidad lógico-interpretativa y abstractiva en los estudiantes. En la Figura 1 se presenta un contexto de aproximación al *cómo* lograr el desarrollo de esta capacidad, el cual les puede servir a los diseñadores de currículos para tenerlo en cuenta al hacer la integración de contenidos.

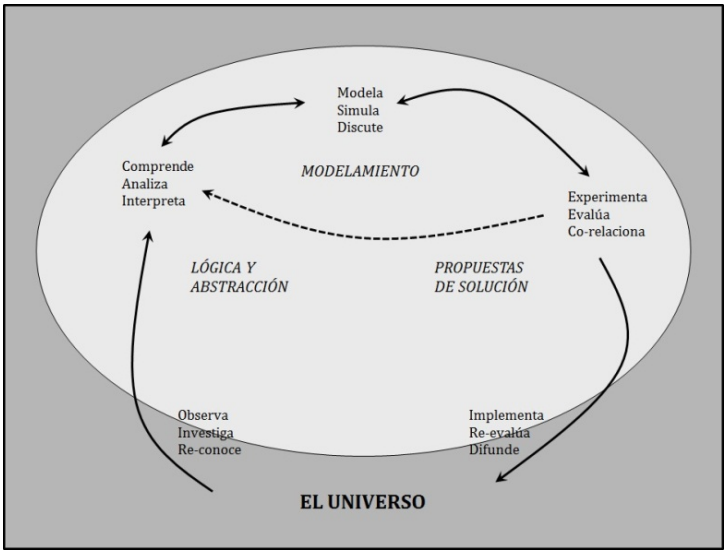

Figura 1. Desarrollo de la capacidad lógico-interpretativa y abstractiva en ingeniería

En esta forma de ver la relación entre el ingeniero y el universo, la mente del profesional es una estructura *sintáctica* mediante la cual observa, comprende, modela, experimenta e implementa su comprensión de éste, y es la forma como se espera que diseñe y presente soluciones a los problemas que investiga o re-conoce. De otro lado, el universo es una estructura *semántica* que incluye al ingeniero mismo y que les da significado a sus pensamientos. Además, es *dinámica*, porque cambia continuamente y solo existe en el aquí y el ahora. Sin embargo, el profesional puede *grabar* sus experiencias cambiantes en la mente y formular soluciones generales, y a continuación puede utilizar las soluciones, que explican los resultados experimentales, para alcanzar el objetivo de solucionar la situación problemática previamente re-conocida.

El ingeniero observa los acontecimientos que tienen lugar en el universo junto con las propiedades que los inician y terminan, y utiliza su capacidad lógico-interpretativa y abstractiva para derivar conclusiones de esas observaciones. En muchos casos, esas conclusiones son acciones instintivas provocadas por asociaciones estímulo-respuesta, que también se pueden expresar en la forma lógica de simulaciones. Posteriormente, ejecuta acciones para experimentar el modelamiento y re-evaluar los resultados para mejorar sus conclusiones, previo a presentar y difundir nuevamente en el universo el modelo de una solución que procesó a través y desde la comprensión del contexto.

Los resultados de esta secuencia de acciones pueden depender no solo de las propias acciones de los ingenieros, sino también de las acciones de otros agentes o condiciones que están por fuera de su control. Aquí cobra importancia una formación estructurada en lógica y abstracción, porque el ingeniero puede no ser capaz de determinar con certeza si esas condiciones se cumplen, pero sí ser capaz de juzgar la posibilidad o probabilidad de ocurrencia. Para lograrlo necesita utilizar las técnicas de la teoría de decisiones, para combinar sus juicios de probabilidad y utilidad, y elegir el curso de acción que le ofrezca mayor posibilidad de éxito. Entre los criterios que puede utilizar para decidir entre las alternativas para el cumplimiento de sus objetivos tiene a su disposición la experiencia vivida por otros ingenieros en el *universo*. Aplicando su capacidad lógica para combinar las diferentes alternativas podrá comprender y apreciar sus propias experiencias, metas y creencias, y combinarlas con las de los demás, de esta manera podrá evitar el conflicto de soluciones repetidas sobre resultados iguales y aportar el conocimiento adquirido para enriquecer el conocimiento acumulado en el universo (disciplina).

5. CONCLUSIONES

Uno de los objetivos de los procesos formativos es preparar a los estudiantes para adaptarse a nuevas y cambiantes situaciones problemáticas. Algunos enfoques pedagógicos buscan equivalencias, aunque su única métrica de aprendizaje sea medir la capacidad que tienen los estudiantes para memorizar la información que se les presenta de forma concreta. Las diferencias formativas se hacen más evidentes al evaluarlas desde la perspectiva de qué tan bien se transfiere ese conocimiento a la solución de los nuevos problemas y configuraciones, es decir, a los que se enfrentan en la vida profesional. Algunas características importantes de esos procesos afectan la capacidad del estudiante para transferir lo que aprende. Una de ellas es la cantidad y tipo de aprendizaje inicial (estrato cultural), con el que desarrolla experiencia y capacidad para transferir conocimiento.

Los estudiantes de ingeniería se inclinan por los temas complejos y la resolución de problemas, siempre y cuando sean interesantes para ellos, además y en teoría, por la oportunidad de utilizar sus conocimientos para crear productos y beneficios para los demás. Pero si los problemas no les llaman la atención, se incrementa la desmotivación y decae su capacidad para comprenderlos y,

mucho más, para presentarles solución. Algunos investigadores proponen una formación utilizando estudios de caso lo más cercanos a la vida cotidiana, porque de esta forma el estudiante desarrolla: 1) su capacidad lógica, para comprenderlos, y 2) su capacidad de abstracción, para modelarlos. Por otro lado, el contexto en el que el ingeniero se forma es un agente importante para lograr la transferencia de conocimiento. De acuerdo con los resultados de la revisión a la literatura es menos probable que un estudiante, que se forma en un único contexto, logre el objetivo del curso, lo que puede variar cuando está expuesto a múltiples contextos. De esta manera tiene mayores posibilidades para comprender y abstraer las características relevantes del problema y para desarrollar una representación más flexible del mismo.

Si el objetivo de formar es ofrecerle a la sociedad profesionales que generen confianza, lo recomendable sería modificar los procesos formativos a los cuales están expuestos. La sociedad de este siglo necesita profesionales confiables desde lo ético y lo humano, pero fundamentalmente solucionadores de problemas. La sociedad crea, desarrolla y convive con problemas que exigen soluciones eficientes y eficaces, pero de acuerdo con las tendencias actuales los nuevos profesionales no lo están logrando. Aunque hasta el momento no es posible responder si el desarrollo de la capacidad lógico-interpretativa y abstractiva es una cuestión genética o que se adquiere a través de procesos formativos, sí se puede asegurar que los modelos pedagógicos, los planes de estudios y los programas actuales no están estructurados integralmente para desarrollarla suficientemente, por lo que se necesita cambios importantes en cada uno de ellos [60].

Otra recomendación es atender y estructurar soluciones formativas de acuerdo con las necesidades de cada contexto, porque, como lo evidencian los estudios neuro-científicos, cada persona es un universo que tiene diferentes motivaciones formativas y diferentes ritmos de aprendizaje, y que se desenvuelve en diferentes ambientes del mismo. Por todo esto se necesita procesos e iniciativas para desarrollar su capacidad lógico-interpretativa y abstractiva, de tal forma que la pueda adaptar y utilizar de acuerdo con sus exigencias y necesidades individuales.

REFERENCIAS

[1] Popper K. (1999). All Life Is Problem Solving. Routledge.
[2] Gagné R. (1960). Conditions of Learning. Holt Rinehart and Winston.
[3] Newell A. y Simon H. (1972). Human Problem Solving. Prentice-Hall.
[4] Bransford J. y Stein B. (1993). The IDEAL Problem Solver: A Guide for Improving Thinking, Learning, and Creativity. Freeman.
[5] Gick M. (1986). Problem-Solving Strategies. Educational Psychologist 21(1-2), 99-120.
[6] Smith M. (1990). A View from Biology. In Smith M.U. (Ed.), Toward a Unified Theory of Problem Solving. Routledge.
[7] Piaget J. (1957). Logic and psychology. Basic Books.
[8] Inhelder B. y Matalon B. (1960). The study of problem solving and thinking. En P. Mussen (Ed.), Handbook of research methods in child development (pp. 421-455). Wiley.
[9] Hill S. (1961). A study of the logical abilities of children. Doctoral dissertation. Stanford University.
[10] McLaughlin G. (1963). PsychOlogic: A possible alternative to Piaget's formulation. British Journal of Educational Psychology 33(1), 61-67.
[11] Inhelder B. y Piaget J. (1964). The early growth of logic in the child: Classification and seriation. Harper and Row.
[12] Furth H. y Youniss J. (1965). The influence of language and experience on discovery and use of logical symbols. British Journal of Psychology 56(4), 381-390.
[13] Suppes P. (1965). On the behavioral foundations of mathematical concepts. Monographs of the Society for Research in Child Development 30(1), 60-96.

[14] Youniss J. y Furth M. (1964). Attainment and transfer of logical connectives in children. Journal of Educational Psychology 55(6), 357-361.
[15] Youniss J. y Furth M. (1967). Learning of logical connectives by adolescents with single and multiple instances. Journal of Educational Psychology 58(4), 222-230.
[16] Suppes P. y Feldman S. (1969). Young Children's Comprehension of Logical Connectives. Journal of Experimental Child Psychology 12(3), 304-317.
[17] McCarthy J. y Hayes P. (1969). Some philosophical problems from the standpoint of AI. En B. Meltzer y Michie D. (Eds.), Machine Intelligence 4. Edinburgh University Press.
[18] Sloman A. (1971). Interactions between philosophy and artificial intelligence: The role of intuition and non-logical reasoning in intelligence. Artificial Intelligence 2(3-4), 209-225.
[19] Boroditsky L. y Ramscar M. (2002). The roles of body and mind in abstract thought. Psychological Science 13(2), 185-189.
[20] Piaget J. (1972). The psychology of the child. Basic Books.
[21] Schank R. (1982). Dynamic Memory. Cambridge University Press.
[22] Kolodner J. (1984). Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model. Lawrence Erlbaum Associates.
[23] Ross B. (1984). Remindings and Their Effects in Learning a Cognitive Skill. Cogn. Psyc. 16(3), 371-416.
[24] Kolodner J. et al. (1985). A process model of cased-based reasoning in problem solving. En 9th international joint conference on Artificial intelligence. Los Angeles, USA.
[25] O'Rorke P. (1988). Automated Abduction and Machine Learning. En Symposium on Explanation-Based Learning. USA, Stanford.
[26] Dasigi V. y Reggia J. (1989). Parsimonious Covering as a Method for Natural Language Interfaces to Expert Systems. Artificial Intelligence in Medicine 1(1), 49-60.
[27] Peng Y. y Reggia J. (1990). Abductive Inference Models for Diagnostic Problem-Solving. Springer.
[28] Kumar G. y Venkataram P. (1994). A realistic model for diagnostic problem solving using abductive reasoning based on parsimonious covering principle. En Third Turkish Conference on Artificial Intelligence and Neural Networks. Ankara, Turkey.
[29] Gottinger H. y Weimann H. (1995). Intelligent inference systems based on influence diagrams. Decision Support Systems 15(1), 27-43.
[30] Jarke M. y Radermacher F. (1988). The AI Potential of Model Management and its Central Role in Decision Support. Decision Support Systems 4(4), 387-404.
[31] Gottinger H. y Weimann H. (1992). Intelligent Decision Support Systems. Decision Support Systems 8(4), 317-332.
[32] Morris A. y Sloutsky V. (1998). Understanding of logical necessity: Developmental antecedents and cognitive consequences. Child Development 69(3), 721-741.
[33] Pietarinen A. (2003). What do epistemic logic and cognitive science have to do with each other? Cognitive Systems Research 4(3), 169-190.
[34] Huitt W. y Hummel J. (2003). Piaget's theory of cognitive development: Educational Psychology Interactive. Valdosta State University.
[35] Pietarinen A. (2004). Logic, Neuroscience and Phenomenology: In Cahoots? En First International Workshop on Philosophy and Informatics. Cologne, Germany.
[36] Egorov I. (2007). Neural logic molecular, counter-intuitive. Biomolecular Engineering 24(3), 293-299.
[37] McLearn G. et al. (1997). Substantial genetic influence on cognitive abilities in twins 80 or more years old. Science 276(5318), 1560-1563.
[38] Winterer G. y Goldman D. (2003). Genetics of human prefrontal function. Brain Research Reviews 43(1), 134-163.
[39] Oldham M. et al. (2006). Conservation and evolution of gene coexpression networks in human and chimpanzee brains. PNAS 103(47), 17973-17978.
[40] Popesco M. et al. (2006). Human lineage-specific amplification, selection, and neuronal expression of DUF1220 domains. Science 313(5791), 1304-1307.
[41] Reuter M. et al. (2006). Identification of first candidate genes for creativity: a pilot study. Brain Research Reviews 1069(1), 190-197.
[42] Lieberman M. et al. (2002). Reflection and Reflexion: A Social Cognitive Neuroscience Approach to Attributional Inference. Advances in Experimental Social Psychology 34, 199-249.

[43] DeWall C. et al. (2008). Evidence that logical reasoning depends on conscious processing. Consciousness and Cognition 17(3), 628-645.
[44] Serna E. (2013). Logic in Computer Science. Revista de Educación en Ingeniería 8(15), 62-68.
[45] Bouhnik D. y Giat Y. (2009). Teaching High School Students Applied Logical Reasoning. Journal of Information Technology Education 8, 1-16.
[46] Simon M. (1996). Beyond inductive and deductive reasoning: The search for a sense of knowing. Educational Studies in Mathematics 30(2), 197-210.
[47] Harel G. y Sowder L. (1998). Students' proof schemes: Results from exploratory studies. Research in collegiate mathematics education 3, 234-283.
[48] Lithner J. (2000). Mathematical reasoning in school tasks. Educational Studies in Mathematics 41(2), 165-190.
[49] Cañadas M. et al. (2009). Using a Model to Describe Students' Inductive Reasoning in Problem Solving. Electronic Journal of Research in Educational Psychology 7(17), 261-278.
[50] Ibañes M. (2001). Cognitive aspects of learning mathematical proofs in students in fifth year of secondary education. Doctoral dissertation. Universidad de Valladolid.
[51] Marrades R. y Gutiérrez A. (2000). Proofs produced by secondary school students learning geometry in a dynamic computer environment. Educational Studies in Mathematics 44(1-3), 87-125.
[52] Stenning K. y Monaghan P. (2005). Strategies and knowledge representation. En J. Leighton y R. Sternberg (Eds.), The nature of reasoning (pp. 129-168). Cambridge University Press.
[53] Halpern J. y Pucella R. (2011). Dealing with logical omniscience: Expressiveness and pragmatics. Artificial Intelligence 175(1), 220-235.
[54] Eberle R. (1974). A logic of believing, knowing and inferring. Synthese 26(3-4), 356-382.
[55] Moore R. y Hendrix G. (1979). Computational models of beliefs and the semantics of belief sentences. Technical Note 187. SRI International.
[56] Konolige K. (1986). A Deduction Model of Belief. Morgan Kaufmann.
[57] Fagin R. y Halpern J. (1987). Belief, awareness, and limited reasoning. Artificial Intelligence 34(1), 39-76.
[58] Halpern J. et al. (1994). Algorithmic knowledge. En 5th Conference on Theoretical Aspects of Reasoning about Knowledge. San Jose, USA.
[59] Rantala V. (1982). Impossible world's semantics and logical omniscience. Acta Philosophica Fennica 35, 18-24.
[60] Serna E. (2011). Abstraction as a critical component in Computer Science training. Avances en Sistemas e Informática 8(3), 79-83.
[61] Bennedsen J. y Caspersen M. (2006). Abstraction ability as an indicator of success for learning object-oriented programming? SIGCSE Bulletin 38(2), 39-43.
[62] Armoni M. y Gal-Ezer J. (2007). Non-determinism: An abstract concept in computer science studies. Computer Science Education 17(4), 243-262.
[63] Andrews P. (2002). An Introduction to Mathematical Logic and Type Theory: To Truth through Proof. Springer.
[64] Gibbs R. (1994). The poetics of mind: Figurative thought, language, and understanding. Cambridge University Press.
[65] Piaget J. e Inhelder B. (1969). The Psychology of the Child. Routledge & Kegan Paul.
[66] Huitt W. y Hummel J. (2003). Piaget's theory of cognitive development. Valdosta State University.
[67] Kuhn D. et al. (1977). The development of formal operations in logical and moral judgment. Genetic Psychology Monographs 95(1), 97-188.
[68] Hall E. (1976). Beyond Culture. Anchor Books.
[69] Wason P. y Johnson P. (1972). Psychology of Reasoning: Structure and Content. B.T. Batsford LTD.

CAPÍTULO VIII
El razonamiento lógico como requisito funcional en ingeniería[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

De forma generalizada se acepta que el trabajo de los ingenieros consiste fundamentalmente en detectar, reconocer y resolver problemas, pero la mayoría de *sistemas educativos* y contenidos temáticos relacionados parece desconocer la necesidad de formar a los estudiantes para que desarrollen razonamiento lógico, para que puedan cumplir apropiadamente con esa función. En este capítulo se hace un recorrido por los conceptos de lógica, abstracción, resolución de problemas y razonamiento lógico, los cuales se analizan y describen como una necesidad funcional de la ingeniería y de su ejercicio profesional, teniendo en cuenta las exigencias de la sociedad del siglo XXI, y haciendo una relación ajustada a los procesos formativos de los actuales y futuros ingenieros para el nuevo orden mundial.

[1] Publicado en inglés en International Journal of Computer Theory and Engineering 7(4), 325-331. 2015.

INTRODUCCIÓN

El trabajo de los ingenieros consiste principalmente en detectar, reconocer y resolver problemas, pero en este siglo, cuando la evolución social ha llevado a la sociedad a las puertas de un nuevo orden mundial, esta función también se ha convertido en parte integral de la labor de la mayoría de profesionales. Los seres humanos conviven en medio de problemas, desde los más simples hasta los más complejos, y cuando se reúnen en conglomerados sociales incrementan su complejidad. Esta sociedad, más que nunca antes en la historia, enfrenta complicados desafíos que debe comprender, analizar y solucionar para asegurar su supervivencia, y proyectar la de la siguiente [1].

Para atender este requerimiento los sistemas de formación deben mantener una continua comunicación con la realidad, con el objetivo de preparar a los futuros profesionales para que se desempeñen adecuadamente cuanto les corresponda vivirla. Este objetivo tiene una característica básica: *la necesidad de desarrollar un pensamiento lógico y una adecuada interpretación abstracta,* para lograr la resolución eficiente y efectiva de esos problemas. En la formación de ingenieros para el siglo XXI esta necesidad es un componente básico, porque su desempeño estará regido ampliamente por una adecuada interpretación del problema, antes que de presentarle una solución.

La práctica ingenieril se puede describir como la solución óptima y práctica de problemas físicos, a través del análisis lógico, sistemático e integral de los hechos científicos. Sin embargo, el número, la complejidad y la falta de claridad de los mismos es tal, que para lograrlo se debe adicionar el juicio y la invariabilidad. Estos componentes hacen parte activa de la intuición personal, la cual se considera como un arte relacionado enteramente con la lógica y la abstracción. El juicio es un reconocido componente de la práctica de la ingeniería, porque la eficiencia y efectividad de las soluciones que se proponen, también dependen de una serie de factores intangibles.

La ingeniería es un campo de las ciencias aplicadas que descansa sobre las bases de la matemática, la física y la química. Para lograr que sus productos respondan a las necesidades sociales, los profesionales en esta área deben adquirir una comprensión amplia y funcional de los procesos, además de un adecuado dominio de las habilidades técnicas [2, 3]. Entre otras habilidades, deben desarrollar una comprensión profunda de los conceptos abstractos, capacidad de pensamiento algorítmico y un razonamiento lógico adecuado [4, 5].

Diversos estudios indican que la habilidad de razonamiento lógico no es independiente de la capacidad intelectual general, y que los estudiantes que razonan lógicamente y resuelven adecuadamente los problemas tienden a obtener mejores resultados en cualquier materia científica [6, 7]. Por lo tanto, la formación en ingeniería, como área científica, debe incluir a la lógica, la abstracción, la matemática y la resolución de problemas en todos los niveles; además, porque como profesionales se espera que dominen y apliquen adecuadamente el pensamiento lógico. Paradójicamente, pocos programas en el mundo atienden esta necesidad formativa [3].

Los ingenieros deben desarrollar la capacidad lógico-interpretativa y abstractiva para desarrollar ese pensamiento, porque su objetivo formativo, al igual que el de los científicos, es ser lógicos y sistemáticos en su razonamiento. Sin embargo, nuevamente, casi ningún *modelo de enseñanza* incluye estas cuestiones en sus procesos. El éxito de la ingeniería del siglo XXI depende, en gran medida, de que los estudiantes hayan convivido desde los primeros niveles con la lógica y el razonamiento lógico, para que puedan potencializarlos y aplicarlos adecuadamente. Desarrollar esta capacidad no es un proceso de último momento, antes de ser profesional; el proceso debe

iniciar desde la escuela e ir madurando, en la misma medida que se incrementa el nivel de formación y la exigencia de los problemas.

1. LA LÓGICA EN LA FORMACIÓN INGENIERIL

Actualmente, en la mayoría de instituciones los procesos formativos están sobrecargados de información, que los profesores reproducen en el aula, generalmente desde un texto determinado. Esto no aporta al objetivo formacional de desarrollar un razonamiento lógico en los estudiantes, porque *aprenden*, o mejor se *saturan*, con una cantidad de fórmulas y conceptos cuya aplicabilidad es casi inexistente. De esta forma se capacitan para resolver tareas repetitivas y problemas simples, pero no desarrollan un razonamiento para solucionar lógicamente problemas con algún grado de complejidad, porque este tipo de contextos no se trabaja en el aula, aunque se tenga los conocimientos necesarios para hacerlo. Por ejemplo, un estudiante de ingeniería asiste a varios cursos de matemática en su proceso, pero debido a que el currículo no está integrado, las fórmulas y conceptos que *aprende* no tienen ningún valor, cuando debe resolver problemas que necesitan integración de éstas y otras áreas del programa [8].

Entonces, se requiere procesos formativos que involucren a la lógica de forma paralela e integradora a lo largo y ancho del currículo. En ingeniería, el conocimiento sin aplicación práctica no es más que, como lo llaman los mismos estudiantes, *relleno* académico para complementar los créditos. Los problemas de este siglo necesitan soluciones innovadoras, creativas y retadoras, pero esto solo será posible en la medida que se *enseñe* lógica de manera diferente y se aproveche el razonamiento lógico que, por naturaleza, deben poseer los estudiantes, para potencializarlo mediante procesos experimentales, prácticos e integradores, pero sobre todo que se puedan aplicar en la vida real. Actualmente, esta situación es crítica, pero todavía no desesperada.

Si se modifican un poco los procesos educativos, y a partir de los experimentos de Susanna Epp [9], se podría cambiar radicalmente el nivel de comprensión y de razonamiento en los estudiantes. En su investigación les solicitó no solo resolver los ejercicios propuestos, sino que también les añadieran breves observaciones. Inicialmente esos comentarios eran opiniones para los profesores, porque se relacionaban con que los estudiantes no entendían el material, pero posteriormente se convirtieron en la base para diseñar procesos de interpretación y comprensión, que podían utilizar cuando estudiaban solos. Los resultados fueron alentadores, porque los estudiantes que participaron se convirtieron en críticos lógicos, con su propio trabajo y con el de los compañeros.

Por otro lado, una adecuada formación en lógica permite desarrollar y aplicar procesos de creatividad. Esta relación se puede comprender mediante una comparación entre el funcionamiento del cerebro, durante el proceso de pensamiento, y el de un computador al calcular [10]. Cuando una persona piensa almacena sus recuerdos como información relacionada, de la misma forma que un computador almacena datos en archivos para su posterior recuperación. Cuando necesitan alguna información, tanto el cerebro como el computador, buscan los datos en los archivos almacenados y los ordenan lógicamente para convertirlos en información, y si recopilan nuevos datos los asignan fácilmente al archivo correspondiente.

La cuestión se complica cuando los datos requeridos no se encuentran almacenados, o están corruptos. El computador se limitará a informar que no encuentra información relacionada, o que no puede procesarla por algún error de hardware o software; por el contrario, el cerebro recurre a la capacidad creativa para correlacionar, combinar, mezclar, probar, abstraer y representar los datos desde otros archivos, e intentará responder a la solicitud aplicando un algoritmo como el

que se presenta en la Figura 1. Claro está, para lograrlo la persona necesita haber recibido una adecuada formación en lógica, de lo contrario responderá cual simple computador: *eso no me lo enseñaron*.

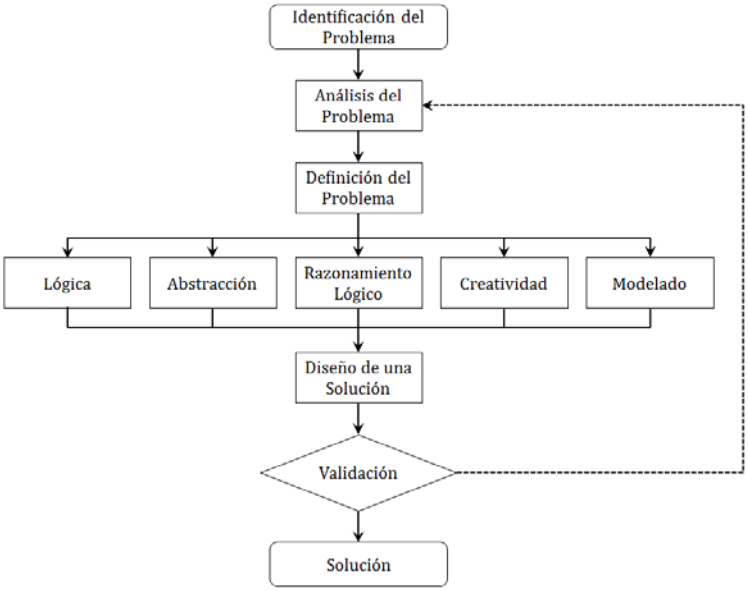

Figura 1. Capacidad creativa del cerebro para solucionar problemas

Resolver problemas lógicamente es un proceso de búsqueda a través de datos conocidos, a los que se adiciona información para complementar el archivo básico en ese tema en particular. Por ejemplo, para resolver un problema matemático el cerebro aplica razonamiento lógico: primero busca en el archivo *cómo aplicar matemáticas,* hasta que encuentre la información relacionada con el problema, pero si no encuentra la información, ese mismo razonamiento lo orienta a hacer ciencia para buscarla y descubrirla, y de esta forma llenará el vacío que tiene en los archivos. Este proceso se logra porque la persona se ha formado en lógica y ha desarrollado un razonamiento que le permite seguir o construir un camino, mediante pasos cuidadosamente estructurados y asegurándose de que cada uno se apoya firmemente en el conocimiento previo [4].

Las habilidades y los principios ingenieriles con los que se forma a los ingenieros se deben orientar a través de procesos lógicamente relacionados, y no puede ser una cuestión de último momento, es decir, en la universidad; es un proceso integral que debe comenzar desde la escuela y que se va desarrollando en la medida que el estudiante progresa en los diferentes niveles. En la formación superior se termina de estructurar a través de aplicaciones más sofisticadas, pero si el estudiante todavía no ha desarrollado esa capacidad, lo más seguro es que opte por abandonar la ingeniería y buscar otra área en la que no sea tan evidente esa falencia [1]. Esto se debe a que gran parte del trabajo de los ingenieros requiere cálculos y análisis, tareas que dependen ampliamente del razonamiento lógico, por lo tanto, estos profesionales deben ser lógicos para tener éxito.

2. APROXIMACIÓN AL RAZONAMIENTO LÓGICO

Se trata de un proceso racional del cerebro a través del cual las personas llegan a conclusiones *correctas*, pero es uno de los conceptos del aprendizaje que suele ser más difícil de alcanzar [11]. Se logra a través del desarrollo de la capacidad lógica y de una relación racional entre los diferentes factores que intervienen en cada situación determinada. El razonamiento lógico depende esencialmente de la habilidad para estructurar y formular procedimientos lógicos y de aplicar procesos inferenciales con un lenguaje preciso [12].

En los resultados de sus investigaciones Smith [13] describe que existe una relación entre los conceptos de razonamiento y pensamiento lógico. Considera al razonamiento como una definición común de pensar, que a veces se utiliza como sinónimo en el significado general de la palabra *pensamiento*. Además, explica que a veces el razonamiento se utiliza para: 1) justificar una conclusión a la que se ha llegado, y 2) convencer a alguien de que acepte una conclusión. Esto implica que el término se podría utilizar para hablar del pasado, como en la primera categoría, o del futuro, como en la segunda categoría. Smith hace hincapié en que los pensamientos o acciones pueden estar fuertemente atados mediante *cadenas de razonamiento*, las cuales hacen las veces de enlaces. Sin embargo, también afirma que la habilidad de razonar viene con la comprensión acerca de lo que se está tratando de razonar, para lo cual se requiere capacidad lógica para interrelacionar ambas cuestiones.

A su vez, Nigel [14] considera que el razonamiento lógico permite la construcción de argumentos eficaces en respuesta a los problemas. Otra cuestión es que las personas, que pertenecen a culturas diferentes, pueden llegar a conclusiones disímiles sobre el mismo asunto, porque razonan diferente [1, 13], pero Smith [13] afirma que esa diferencia no es porque pertenezcan a diferentes culturas y tengan diferentes niveles de habilidad, sino porque tienen diferentes visiones del mundo, en respuesta a como han sido formadas. Porque la cultura es la forma de racionalizar el mundo, pero la capacidad lógico-interpretativa es la forma como se interviene.

Al desempeñar una profesión se puede generar diferencias en el razonamiento lógico [13], por ejemplo, los ingenieros podrían razonar diferente a los médicos y a los filósofos; e incluso dentro de la misma profesión los razonamientos pueden variar, debido a la posición desde la cual se razona. Esto lo investiga Smith [13] y llega a la conclusión de que estas diferencias tienen como base una serie de factores, entre los que se incluye a la cultura, los valores, los roles, las tareas y las personalidades, pero, sobre todo, a la capacidad de comprender los problemas mediante razonamiento lógico. Sin embargo, cuando lo lógico genera conflictos con los valores individuales, usualmente éstos suelen triunfar, porque son más fuertes que la lógica. Aunque esto no suele suceder muy a menudo, porque las personas provienen de diferentes procesos formativos y por tanto razonan diferente. Smith sostiene que lo único que las personas deben desarrollar mejor que la lógica debería ser los valores y el sentido común, pero actualmente parece que la formación se orientara a utilizarlos para anularla.

En este mismo sentido, los valores y el sentido común son consecuencia de la experiencia que acumulan las personas y, por lo tanto, crean inferencias y desacuerdos en el razonamiento lógico. En parte debido a los diferentes puntos de vista que tienen para tomar decisiones y llegar a conclusiones, y porque son el resultado de formulaciones acerca del mundo que no están gobernadas por las reglas de la lógica [13]. Sin embargo, el hecho de aceptar una conclusión es en sí un enfoque de razonamiento. Aunque Smith enfatiza que la lógica no puede ser un principio rector detrás de todas las decisiones y conclusiones acerca de mundo, Baron [15] sugiere que sería más útil si se le diera un papel más relevante en el razonamiento lógico cotidiano, es decir, que lo ideal fuera *poner a funcionar el cerebro antes que la boca*. Esto sería de mucho valor para un ingeniero, porque si se le forma para razonar lógicamente, seguro aplicará el proceso de la Figura 1 antes de atreverse a presentar una solución a los problemas cotidianos.

Por lo tanto, la lógica se debería incluir desde las primeras etapas del proceso formativo de todos los estudiantes. Utilizar sencillas relaciones lógicas para expresar el uso de algunas expresiones simples, que les permita relacionar la lógica con las actividades y los diálogos cotidianos. Convendría estructurar un sistema lógico elemental inmerso en el currículo, de tal manera que los estudiantes comiencen a desarrollar o potencializar su capacidad lógico-interpretativa desde

edad temprana [16]. Pero esta es una cuestión que parece no tener en cuenta los actuales *sistemas de educación*, en los que prima cuestiones como los valores que, aunque importantes y necesarios, no tienen importancia si no se comprenden racionalmente.

Formar en y desarrollar el pensamiento lógico debe ser un objetivo bien planificado. Los ingenieros lo necesitan para tomar decisiones que les permita solucionar problemas sociales, y en este sentido investigadores como Wason y Johnson [16] proponen métodos sofisticados mediante descripciones lógicas, que pueden ser relevantes como simples relaciones lógicas en los diálogos o actividades cotidianas de los estudiantes. Estas propuestas se deberían tener en cuenta, porque animan a la simplicidad y le dan relevancia a la lógica en los contextos cotidianos de los procesos formativos; además, se podrían utilizar para desarrollar el pensamiento lógico en los estudiantes, claro está, a través de actividades que despierten su interés.

2.1 Razonamiento y lógica

La lógica se refiere a la formalización de las leyes del pensamiento y se centra en la formulación de teorías normativas para describir la forma en que las personas deberían pensar. La psicología cognitiva también se ocupa del pensamiento, pero se centra casi exclusivamente en las teorías descriptivas que estudian *cómo* piensan las personas en la práctica, sin importar si es o no correcto. Estas dos teorías se han desarrollado mayoritariamente de forma aislada y sin una relación directa reconocida, sin embargo, en las últimas décadas los psicólogos desarrollaron la teoría de los *procesos duales*, que se comprende como la combinación de teorías descriptivas y normativas [17]. Por su parte, las teorías descriptivas tradicionales se centran en el pensamiento intuitivo, el cual es asociativo, automático, paralelo y subconsciente, mientras que las normativas, por el contrario, se centran en el pensamiento deliberativo, el cual se basa en normas, requiere esfuerzo y es serial y consciente.

De acuerdo a estos principios se puede argumentar que la lógica es una cuestión relacionada con los procesos duales, debido a que combina el pensamiento intuitivo y el deliberativo, pero que no solo se refiere a pensar en abstracto, sino que también simboliza los pensamientos representados en forma de oraciones, y al pensamiento como la manipulación de afirmaciones, para generar nuevos pensamientos. Por lo que la lógica, vista desde esta perspectiva, se puede considerar como la formalización del lenguaje del pensamiento humano [18].

La lógica y el razonamiento son habilidades cognitivas a través de las cuales se llega a conclusiones sólidas, para tomar decisiones y resolver problemas en la vida cotidiana. Las personas llegan a ellas con base en el procesamiento de la información que reciben a través de los sentidos. Pero, a pesar de que constantemente aplican ambas habilidades, muchas veces ni siquiera son conscientes de que lo hacen. Al recibir estímulos y aplicar la lógica para llegar a conclusiones correctas, están realizando la tarea mental de razonamiento, sin embargo, a diferencia de las decisiones instintivas, es decir, las que se realizan sobre la base de respuestas emocionales, el razonamiento cognitivo suele requerir mayores tiempos de respuesta [19].

La lógica y el razonamiento están estrechamente vinculados y, con frecuencia, trabajan juntos para ayudarles a las personas a *funcionar* correctamente, y sin estos dos componentes cognitivos les sería difícil conducir racionalmente sus vidas [18]. Por otro lado, ambos hacen parte y son necesarios en los procesos mentales que los ingenieros aplican para solucionar problemas, y dado que su entorno está constantemente *bombardeado* con estímulos complejos, que involucran numerosas características variables, deben desarrollar la capacidad lógico-interpretativa para responder a estas exigencias. La formación en estos conceptos les ayudará a desarrollar la

habilidad necesaria para enfrentar los entornos dinámicos, es decir, los ingenieros necesitan formación en lógica y en otros conceptos para desarrollar razonamiento lógico para identificar y comprender los problemas, pero también para presentarles soluciones. Cuando logran ese desarrollo adquieren la capacidad de combinar múltiples procesos cognitivos, como la memoria, la atención, la velocidad de procesamiento y la flexibilidad, que necesitan para reconocer patrones, sacar conclusiones y tomar decisiones [20].

2.2 Razonamiento lógico y resolución de problemas

Resolver problemas en ingeniería puede ser divertido, pero también puede ayudar a determinar la dirección de esta carrera, porque en ese proceso los estudiantes deben poner a prueba su lógica y habilidades de razonamiento [21]. Alcanzar un pensamiento crítico fuerte y unas habilidades de razonamiento lógico adecuadas les ayudará a tomar mejores decisiones, y a resolver problemas con mayor eficacia. En cualquier caso, cuando los ingenieros se enfrentan a problemas deben estar lo suficientemente preparados, haber desarrollado su sentido común y habilidades suficientes para diferenciar entre evidencias buenas y malas [22], y ser capaces de extraer conclusiones lógicas a partir de ellas. Entre otras cosas, deben tener en cuenta:

1. Muchas de las cuestiones que ponen a prueba el sentido común se presentan como escenarios de toma de decisiones. Aunque la situación puede ser ajena al ingeniero y las preguntas pueden parecer complicadas, la respuesta la encuentra al recordar cómo dividir el problema en sus partes y cómo pensar lógicamente acerca de la situación. El sentido común es una característica importante que deben desarrollar estos profesionales, y a menudo se representa como un instinto ante una situación a la que responde con lo primero que le ofrezca el razonamiento. Pero conscientemente genera un proceso que le recuerda lo que es correcto y lo que es incorrecto, por lo que es conveniente que aprenda también a escuchar esta parte.

2. Las pruebas de la lógica deductiva miden a menudo las habilidades de razonamiento inductivo, por lo que son útiles para evaluar si una evidencia fuerte de un argumento deductivo es creíble y razonable.

3. Los contextos a los que se enfrentan los ingenieros ponen a prueba sus habilidades de razonamiento, y en algunos casos necesitarán sacar conclusiones desde las evidencias. Una habilidad necesaria para responder a esto es asegurar una respuesta correcta mediante procesos de eliminación. Por lo que, dadas las evidencias que ofrecen los contextos, debe ser capaz de eliminar automáticamente algunas de las posibles respuestas.

3. EL RAZONAMIENTO LÓGICO EN LA INGENIERÍA

En general, la toma de decisiones involucra el mecanismo sensorial, la percepción, la cognición y la expresión de resultados en el cerebro [21], y a menudo se siente, percibe, piensa, recuerda y razona de manera adaptativa, consciente e inconsciente. Por otro lado, cuando los ingenieros se enfrentan a problemas o situaciones en la vida, donde deben tomar una decisión, necesitan aplicar lógica y razonamiento lógico para alcanzar los resultados, por lo que es importante que su formación esté permeada por procesos orientados a desarrollar ambas capacidades.

Generalmente, la lógica se basa en la deducción, un método de inferencia exacta que estudia el razonamiento *correcto* conformado por lenguaje y razonamiento [22]. Por otro lado, el razonamiento lógico implica decidir qué hacer para lograr éxito a partir de la emisión de una intención. Estas decisiones se estructuran mediante un conjunto de acciones viables, un conjunto

de restricciones y un conjunto de posibles caminos a tomar, y la decisión consiste en encontrar la mejor secuencia de eventos, admisibles y aceptables, y de acciones que permitan pasar de la *intención* a la *acción* y, para alcanzar los mejores resultados, cada paso entre el pensamiento y la acción se debe realizar con lógica y razonamiento lógico. Pero a menudo se aplica el sentido común para la toma de decisiones, el cual determina qué hacer, independientemente de lo que se piensa, y es un factor clave para el actuar de los ingenieros, porque, aunque la base de sus decisiones siempre será la lógica, es probable que les ayude a enfrentar la complejidad del mundo real, y les proporcione un acceso directo y rápido a la toma de decisiones críticas.

Otro asunto es cuando estos profesionales se enfrentan con cuestiones morales, ante las cuales muchas veces deben actuar por instinto y detenerse a razonar sobre qué hacer, ante lo cual sus acciones serán más consistentes con sus pensamientos. Frente a circunstancias difíciles o de alta importancia normalmente tienen tres posibilidades para tomar decisiones: 1) que las circunstancias de la situación sean comparables a otras que han enfrentado, 2) que el problema sea diferente de los anteriores, y 3) que algunas cuestiones de la situación ya hayan sido tratadas con éxito. En cualquiera de ellas el ingeniero debe combinar los éxitos previos para alcanzar los resultados deseados y solucionar la situación presente. Pero como no puede ser especialista en cada situación, necesita aplicar razonamiento lógico para proceder [20]. Otros aspectos importantes para el razonamiento lógico son:

- *La lógica filosófica*. Debido a su influencia en la vida de las personas y a sus contribuciones para la solución general de problemas. Entre otras cosas, ayuda a analizar los conceptos, las definiciones, las discusiones y los problemas mismos, y contribuye a la capacidad de organizar ideas y cuestiones relacionadas con cada situación.
- *Las habilidades de comunicación*. Porque de la forma cómo se expresa una persona depende en gran medida que otros comprendan la solución que describe. Las ideas se deben presentar a través de argumentos bien construidos, sistemáticos y razonados.
- *Poder de persuasión*. Para lo que es necesario aprender a construir y defender los puntos de vista propios, y apreciar las posiciones que compiten, e indicar con firmeza porque deber ser considerada como la mejor alternativa.
- *Habilidades de escritura*. Lo que se logra mediante lógica y razonamiento en la realización de escritura interpretativa y argumentativa, retratando detalles de ejemplos concretos.

En este sentido George Boole escribió su obra pensando en investigar las leyes fundamentales de las operaciones de la mente, por las cuales el razonamiento se lleva a cabo, y les dio expresión en el lenguaje simbólico del cálculo, y creía que el razonamiento humano se guiaba por la lógica formal [23]. Esta visión de la lógica se remonta a Aristóteles, quien creó la lógica modal y de silogismos, e introdujo la calificación necesaria y posible a las premisas [24]. Hasta hace poco la comprensión de un sistema eficaz de toma de decisiones se basaba en la lógica formal y la estadística, pero Braverman [23] opinaba que una situación de decisión real, sin importar su nivel de complejidad, se podía descomponer mediante un proceso de reducción en sus partes constitutivas a cualquier nivel de detalle, y que la suma de las soluciones a cada componente individual daría como resultado la solución general, a la que se llega mediante una aplicación continua de razonamiento lógico.

Por otra parte, la hipótesis científica de que las personas tienen lógica inherente como base para el pensamiento racional, fue fuertemente influenciada por los escritos de Piaget e Inhelder [25]; sin embargo, estudios posteriores demostraron que la realidad es diferente. Las falacias lógicas son comunes, y despiertan un interés permanente, debido a que el razonamiento humano es

propenso a ellas [24, 26]. Los hallazgos en varios experimentos de razonamiento lógico demuestran que, a menudo, las personas cometen errores lógicos y sacan conclusiones innecesarias, pero plausibles, con base en sus creencias. La obra de Kahneman et al. [27] respalda la opinión de que el razonamiento lógico viola sistemáticamente las normas para el razonamiento estadístico, ignorando, entre otras cosas, las tasas de base, el tamaño de la muestra y las correlaciones. En su investigación tuvieron en cuenta el razonamiento probabilístico bayesiano, como criterio normativo para que un agente sea perfectamente racional, y encontraron que los humanos están sistemáticamente a la altura de la norma, con lo que concluyeron que, al parecer, el hombre no es un bayesiano conservador: *no es un bayesiano en absoluto*.

Richardson [26] afirma que los estudios de toma decisiones formales deben ser reemplazados por la racionalidad limitada, porque de esta manera se elimina la complejidad de las situaciones del mundo real. Un ingeniero sin formación en lógica tiene dificultades para desarrollar el razonamiento lógico y, por tanto, tendrá dificultades para aislar una tarea específica de razonamiento desde su entorno y se concentrará básicamente en las premisas dadas. Evans [28] afirma que, por defecto, el modo para el razonamiento lógico es pragmático y no deductivo o analítico, y que las personas tienden a seleccionar alternativas creíbles, es decir, a elegir algo que sería plausible en el mundo real, en lugar de seguir las reglas razonadas lógicamente.

El razonamiento lógico, como componente central cognitivo, depende de las teorías de la comprensión, de la memoria, del aprendizaje, de la percepción visual, de la planificación, de la resolución de problemas y de la toma de decisiones [29]. De acuerdo con este autor el cerebro tiene dos caminos complementarios para la toma de decisiones: 1) uno para el razonamiento, y 2) otro para la activación inmediata de experiencias emocionales previas en situaciones semejantes. El segundo es una especie de reacción a una *sensación visceral,* que activa una señal emocional para aumentar la eficiencia del proceso de razonamiento y hacerlo más rápido.

La diferencia en la forma como los ingenieros llevan a cabo estos procesos y como lo hacen otros profesionales es que en la mayoría de situaciones no pueden obrar por instinto, porque no lo pueden considerar como un sustituto para el razonamiento verdadero, aunque éste tome una ruta más larga. Cuando la situación requiere una respuesta el cerebro le pide imágenes relacionadas con la situación y las opciones para la acción, y de esta forma se anticipa a los resultados futuros mediante representaciones abstractas, y a través de estrategias de razonamiento lógico operará sobre ese conocimiento para tomar una decisión.

Naturalmente, la dificultad en el razonamiento lógico para resolver problemas depende de factores diferentes a los mecanismos de razonamiento *per se*. Si un ingeniero no logra comprender el planteamiento del problema, no entenderá la tarea que se supone debe realizar y no podrá estructurar ni formular una solución adecuada, porque sus respuestas no reflejarán un proceso de razonamiento correcto. Varias dificultades metodológicas se observan en estos procesos, pero se ha demostrado que una adecuada formación le permitiría superar esta especie de aislamiento [30].

Para las personas es difícil llevar a cabo algo no-natural, como el razonamiento deductivo, porque tienen que hacer caso omiso de los pensamientos más normales y adherirse a un caso restringido. Los ingenieros, como los científicos, deben estar mejor preparados para pensar en sistemas restringidos y artificiales. Mortimer y Wertsch [10] explican que el lenguaje científico tiene una gramática diferente, y que uno de los problemas para que los estudiantes no logren desarrollar un razonamiento lógico adecuado es porque son reacios al cambio, y no aceptan modificar su lenguaje natural por el discurso teórico utilizado por los profesores. Por otro lado, las habilidades

de argumentación dependen en gran medida de la formación en lógica, porque los sistemas *educativos* parecen concentrarse en diferentes aspectos de la capacidad de análisis, y olvidan lo relacionado con la abstracción y el modelamiento [32].

Por naturaleza, el cerebro humano tiene cierta capacidad para el razonamiento lógico, pero primero necesita potencializar, y en algunos casos desarrollar, la capacidad lógico-interpretativa y abstractiva [33], especialmente en las profesiones cuyo centro de actividad es la resolución de problemas, como la ingeniería. En la Tabla 1 se presenta una adaptación de la taxonomía de Bloom al desarrollo del razonamiento lógico que deben poseer los ingenieros. Lograr el desarrollo de estas habilidades y capacidades debe ser el objetivo de los currículos en ingeniería.

Tabla 1. Taxonomía del desarrollo del razonamiento lógico

Habilidad	Capacidad	Contexto
Conocimiento	Recoger, describir, identificar, listar, mostrar, contar, tabular, definir, examinar, etiquetar nombrar, re-contar, citar, enumerar, partir, leer, registrar, reproducir, copiar, seleccionar.	Fechas, eventos, lugares, vocabulario, ideas clave, partes de un diagrama,…
Comprensión	Asociar, comparar, distinguir, ampliar, interpretar, predecir, diferenciar, contrastar, describir, discutir, estimar, grupo, resumir, ordenar, citar, convertir, explicar, parafrasear, reafirmar, rastrear.	Encontrar significados, transferir e interpretar hechos, inferir causas y consecuencias,…
Aplicación	Aplicar, clasificar, cambiar, ilustrar, resolver, demostrar, calcular, completar, resolver, modificar, mostrar, experimentar, relacionar, descubrir, actuar, administrar, articular, trazar, recoger, computar, construir, determinar, desarrollar, establecer, preparar, producir, reportar, enseñar, transferir, utilizar.	Utilizar información en nuevas situaciones, resolver problemas,…
Análisis	Analizar, organizar, conectar, dividir, inferir, separar, clasificar, comparar, contrastar, explicar, seleccionar, ordenar, desglosar, correlacionar, diagramar, discriminar, enfocar, ilustrar, perfilar, priorizar, subdividir, señalar, priorizar.	Reconocer y explicar patrones y significados, consultar partes y totalidades,…
Síntesis	Combinar, componer, generalizar, modificar, inventar, planear, sustituir, crear, formular, integrar, reorganizar, diseñar, especular, reescribir, adaptar, anticipar, colaborar, compilar, idear, expresar, facilitar, reforzar, estructurar, sustituir, intervenir, negociar, reorganizar, validar.	Discutir *qué pasaría* ante una situación, crear nuevas ideas, predecir y sacar conclusiones,…
Evaluación	Evaluar, comparar, decidir, discriminar, medir, clasificar, probar, convencer, concluir, explicar, graduar, juzgar, resumir, apoyar, evaluar, criticar, defender, persuadir, justificar, replantear.	Hacer recomendaciones, evaluar valores y tomar decisiones, criticar ideas,…
Dominio afectivo	Aceptar, intentar, desafiar, defender, disputar, unir, juzgar, contribuir, alabar, preguntar, accionar, apoyar, colaborar.	Este dominio se refleja en las relaciones interpersonales, las emociones, las actitudes y los valores.

4. CONCLUSIONES

El análisis realizado en este trabajo revela que los estudiantes de ingeniería, a pesar de poseer entrenamiento formal en lógica matemática, frecuentemente aplican un razonamiento pragmático para resolver problemas. La preferencia por este razonamiento plantea ciertas preocupaciones acerca de su capacidad para tomar buenas decisiones en la vida laboral. Debido a que en el razonamiento cotidiano de estos profesionales se requiere estrictos requisitos lógicos, los contenidos curriculares se deberían estructurar con el objetivo de desarrollar en ellos una

lógica diferenciadora, porque en las decisiones profesionales de ingeniería es necesario respetar una serie de reglas lógicas.

La necesidad del razonamiento lógico en la vida profesional de los ingenieros lleva a la conclusión de que en sus procesos formativos se debe enfatizar en la lógica y el pensamiento sistemático. Los ingenieros deben ser capaces de seleccionar adecuadamente un razonamiento lógico para cada situación, y de alternar entre el razonamiento cotidiano, el formal y riguroso, y la solución creativa y heurística de los problemas. Por todo esto es necesario potencializar en ellos una buena capacidad para reflexionar acerca de las funciones cognitivas y de las habilidades meta-cognitivas. Por lo tanto, el objetivo de desarrollar la capacidad lógico-interpretativa y abstractiva necesita ser abordado explícitamente en los planes de estudios.

Este análisis también indica que el lenguaje afecta más de lo esperado la capacidad de razonamiento lógico formal de los ingenieros. El resultado sugiere que el lenguaje, como medio de estudio, tiene un efecto más fuerte en el aprendizaje de la ciencia y la ingeniería de lo que se cree comúnmente. Si este hallazgo se confirma con estudios posteriores se deberá prestar mayor atención a la forma como se capacita en lectura y escritura en general. Sin embargo, se necesita más estudios para confirmar y explicar en qué medida las influencias lingüísticas influyen el desarrollo del razonamiento lógico.

La lógica y el razonamiento lógico son importantes en la formación y el desarrollo profesional de los ingenieros. En ninguna otra área del conocimiento es tan necesaria este tipo de formación, porque a través de su adecuado desarrollo serán capaces de ampliar la gama de cosas que conocen y comprenden, de propiciar el auto-conocimiento, de comprender los problemas y de presentar soluciones eficientes y eficaces. Por lo tanto, los sistemas educativos y los programas curriculares deberán darle la importancia que se merecen estas áreas, e incluirlas relacionalmente en los currículos. De esta manera será posible que los futuros ingenieros puedan resolver adecuadamente los complicados problemas de la sociedad del siglo XXI en el nuevo orden mundial.

REFERENCIAS

[1] Gellatly A. (1986). Skill at reasoning. En A. Gellatly A. (Ed.), The Skilful Mind: An Introduction to Cognitive Psychology (pp. 159-170). Open University Press.

[2] Hakkarainen K. et al. (2004). Communities of Networked Expertise. Professional and Educational Perspectives. Elsevier.

[3] Moss J. et al. (2006). The role of functionality in the mental representations of engineering students: Some differences in the early stages of expertise. Cogn. Sci. 30, 65-93.

[4] Eckerdal A. y Berglund A. (2005). What does it take to learn 'Programming thinking'? En 1st Int. Computing Education ResearchWorkshop. Seattle, USA.

[5] Faux R. (2006). Impact of preprogramming course curriculum on learning in the first programming course. IEEE Trans. Educ. 49(1), 11-16.

[6] Johnson M. y Lawson A. (1998). What are the relative effects of reasoning ability and prior knowledge on biology achievement in expository and inquiry classes? Res. Sci. Teach. 35, 89-103.

[7] Capizzo M. et al. (2006). The impact of the pre-instructional cognitive profile on learning gain and final exam of physics courses: A case study. Eur. J. Eng. Educ. 31, 717-727.

[8] Pólya G. (1948). How to solve it, A new aspect of mathematical method. Princeton University Press.

[9] Epp S. (1996). A Cognitive Approach to Teaching Logic and Proof. En Symposium on Teaching Logic and Reasoning in an Illogical World. Piscataway, USA.

[10] Gazzaniga M. (2002). The split brain revisited. Sci. Amer. 12, 27-31.

[11] Haygood R. y Bourne L. (1968). Attribute and rule-learning aspects of conceptual behavior. En P. Wason y P. Johnson (Eds.), Thinking and Reasoning (pp. 234-259). Penguin Books.

[12] Carrol J. (1964). Language and Thought. Prentice-Hall.
[13] Smith F. (1992). To Think. Routledge.
[14] Nigel B. (2002). Dovetailing language and context: Teaching balanced argument in legal problem answer writing. English for Specific Purposes 21(4), 321-345.
[15] Baron J. (1994). Thinking and Deciding. CUP.
[16] Wason P. y Johnson P. (1972). Psychology Of Reasoning: Structure and Content. B.T. Batsford LTD.
[17] Basak C. et al. (2008). Can training in a real-time strategy video game attenuate cognitive decline in older adults? Psychology and Aging 23(4), 765-77.
[18] Dahlin E. et al. (2008). Plasticity of executive functioning in young and older adults: Immediate training gains, transfer, and long-term maintenance. Psychology and Aging 23(4), 720-30.
[19] DeWall C. et al. (2008). Evidence that logical reasoning depends on conscious processing. Consciousness and Cognition 17(3), 628-45.
[20] Sorel O. y Pennequin V. (2008). Aging of the Planning process: The role of executive functioning. Brain and Cognition 66(2), 196-201.
[21] Hall E. (1976). Beyond Culture. Doubleday.
[22] Kline S. y Delia J. (1990). Reasoning about communication and communicative skill. En B. Jones y L. Idol (Eds.), Dimensions of Thinking And Cognitive Instruction (pp. 177-209). Lawrence Erlbaum Associates.
[23] Braverman J. (1972). Probability, Logic, and Management Decisions. McGraw-Hill.
[24] Hintikka J. (1996). Inquiry as inquiry: A logic of scientific discovery. Jaakko Hintikka Selected Papers.
[25] Piaget J. e Inhelder B. (1969). The Psychology of the Child. Basic Books.
[26] Richardson R. (1998). Heuristics and satisficing. En W. Bechtel y G. Graham (Eds.), A Companion to Cognitive Science (pp. 566-575). Blackwell.
[27] Kahneman D. et al. (1982). Judgment Under Uncertainty: Heuristics and Biases. Cambridge Univ. Press.
[28] Evans J. (2002). Logic and human reasoning: An assessment of the deduction paradigm. Psych. Bul. 128, 978-996.
[29] Damasio A. (2003). Looking for Spinoza. Joy, Sorrow and the Feeling Brain. Vintage/Random House.
[30] Rips L. (1998). Reasoning. En W. Bechtel y G. Graham (Eds.), A Companion to Cognitive Science (pp. 299-305). Blackwell.
[31] Mortimer E. y Wertsch, J. (2003). The architecture and dynamics of intersubjectivity in science classrooms. Mind. Culture. Act. 10, 230-244.
[32] Marttunen M. et al. (2003). Comparison of argumentation skills among secondary school students in Finland, France and the United Kingdom. En United Nations Educational, Scientific and Cultural Organization Conf. Intercultural Education. Jyväskylä, Finland.
[33] Tenopir C. y King D. (2004). Communication Patterns of Engineers. IEEE Press.

CAPÍTULO IX
Proceso y progreso de la formalización de requisitos en Ingeniería del Software[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

Desde mediados del siglo XX se inició la investigación en métodos formales y se presentaron propuestas y metodologías para aplicarlos en el desarrollo de software. La idea era superar la diagnosticada crisis del software mediante la matematización del ciclo de vida del desarrollo de este producto. En este capítulo se presenta los resultados de una revisión de la literatura acerca del progreso y desarrollo de la formalización de requisitos. La conclusión es que ambos aspectos son lentos: no hay interés suficiente en la industria, la academia no capacita en métodos formales, no hay patrocinio para este campo de investigación, los profesionales le temen a las matemáticas y las metodologías tradicionales de la Ingeniería del Software siguen siendo las más utilizadas por los equipos de trabajo. Debido a la deficiencia en la calidad, la seguridad y la fiabilidad del software, es necesario potencializar la investigación y experimentación con los métodos formales, porque la esperanza es que las matemáticas serán la herramienta con la que se supere la crisis del software promulgada en los años 60.

[1] Publicado en inglés en Ingeniare. Revista chilena de ingeniería 28(3), 411-423. 2020.

INTRODUCCIÓN

Las empresas se enfrentan a un entorno cambiante en el que las demandas y soluciones para los nuevos productos tienen que ser analizadas y evaluadas cuidadosamente. Con el fin de mantener la competitividad, deben elegir una combinación adecuada de funciones y tecnologías, y ser capaces de ponerlas en el mercado de la forma más económica posible. Con ciclos de vida más cortos se ven obligadas a llevar los productos aún más rápido a la madurez del mercado [1]. Además, el tiempo o la planificación de horizontes para los nuevos productos tiene que sincronizarse con cuidado. De esta manera, es posible desarrollar y alinear las soluciones a las necesidades de los usuarios.

Cuando se diseña un sistema se recomienda que los objetivos, las funcionalidades y las restricciones se identifiquen con la mayor precisión posible. Esto es lo que constituye el documento de requisitos o la especificación de los mismos. Algo que se repite cotidianamente en este proceso es que los requisitos suelen ser escritos en lenguaje natural, ya sea porque los ingenieros no conocen los lenguajes formales o porque consideran que es demasiado pronto en el ciclo de vida como para utilizar una especificación formal, es decir, aducen no tener suficientes datos para escribir con éxito un requisito formal. El principal problema con la escritura de requisitos en lenguaje natural es que no pueden servir como insumos para las técnicas de verificación y validación automatizadas.

De hecho, si en el plan de pruebas se pretende utilizar técnicas automatizadas de verificación, los lenguajes formales son obligatorios, porque son los únicos lenguajes que pueden entender los computadores. Por otra parte, no son ambiguos, es decir, que una sentencia no puede ser entendida de diferentes maneras. Esto implica que los seres humanos también se pueden beneficiar del uso de estos lenguajes, porque entenderán plenamente los requisitos sin una interpretación previa del usuario. El principal problema es que ambas actividades, la lectura y la escritura, no son fáciles de lograr cuando se trata de lenguajes formales.

De acuerdo con la práctica industrial tradicional los requisitos se especifican en lenguaje natural, y la comprobación de errores de forma manual. Las deficiencias de esta cadena de herramientas son bien conocidas: ambigüedad en la descripción de los requisitos, incremento en costos y tiempos de entrega, y procesos de reingeniería en el diseño y la arquitectura [2]. Además, las revisiones pueden detectar errores, pero no garantizar su ausencia. Una manera de evitar estos inconvenientes es a través de la formalización de requisitos y el análisis formal automático. Diversas investigaciones se han llevado a cabo con este objetivo, especialmente con soporte desde los lenguajes formales y las herramientas de automatización [3-8]. Pero, si bien es posible determinar si estos resultados son factibles en la práctica, no se puede lograr sin un decidido argumento de principios que se apliquen uniformemente en proyectos de la vida real.

El proceso de formalización consiste en escribir requisitos utilizando una notación formal matemática. Lo que puede ser una tarea difícil si se hace directamente desde un requisito informal y sin una verdadera guía para la formalización: solamente los conocimientos y las experiencias, y, tal vez, algunas heurísticas [9] pueden ayudar a saber cómo conseguir una escritura formal. Los enfoques formales pueden ayudar con un análisis profundo a través de semánticas precisas de los requisitos, sin embargo, la formalización es una tarea difícil y a menudo se utiliza separada de la validación. El método tradicional para resolver este problema es conformar un equipo con conocimiento y experiencia en formalización y de los diferentes componentes de la solución planificada. Pero incluso estos equipos tienen que recurrir a materiales de entrenamiento, libros, de ayuda del sistema, escenarios de uso y otro tipo de documentos que describan los procesos

del negocio y los enfoques de configuración. El problema es que el análisis de esa documentación toma tiempo, porque contiene información no-estructurada y no-formalizada.

Las estadísticas muestran la importancia de seleccionar un enfoque adecuado de Ingeniería de Requisitos para el desarrollo del proyecto. Los estudios realizados por Boehm [10, 11] y otros han demostrado que el impacto potencial de una formulación deficiente de requisitos es sustancial. Boehm sugiere que los errores en la especificación y el análisis de requisitos son los más numerosos en un sistema, y que la mayoría de ellos no se encuentra en la etapa de desarrollo, sino en las pruebas y la entrega. El costo resultante para corregir estos errores tiene una relación directa con el tiempo invertido en su búsqueda. Un error que se encuentre en la etapa de requisitos cuesta alrededor de una quinta parte de lo que sería si se encuentra en la fase de pruebas, y una quinceava parte de lo que costaría después de que el sistema esté en uso. De ahí la importancia de trabajar en la formalización de los requisitos, porque de esta manera se podría entregar requisitos no-ambiguos y posiblemente con pocos errores.

En este capítulo se presenta un análisis al progreso y proceso de la formalización de requisitos. El objetivo es trazar un mapa acerca de qué se ha hecho, qué se ha logrado y cuál es el futuro de la formalización como herramienta en la Ingeniería de Requisitos.

1. TRABAJOS RELACIONADOS

Mathiassen y Munk [12] afirman que las formalizaciones se relacionan tanto con los tipos de expresiones como con los de comportamiento. En su trabajo discuten los límites para aplicar la formalización en ambos sentidos y los ilustran con ejemplos prácticos de desarrollo de sistemas. Además, establecen que las formalizaciones son valiosas en algunas situaciones, pero insuficientes en otras. La alternativa es usarlas de manera acrítica y analizar cada situación particular, y de ahí elegir una combinación entre un enfoque formal y uno informal. No hacen un análisis al desarrollo de la formalización hasta el momento, ni tampoco proyectan su uso futuro.

Fantechi y sus colegas [13] presentan un prototipo para la formalización de requisitos en el diseño de sistemas reactivos. Es una herramienta para traducir automáticamente frases en lenguaje natural a fórmulas de lógica temporal. De acuerdo con ellos las frases en lenguaje natural se utilizan para expresar requisitos de manera informal, y que la lógica temporal es adecuada para expresar las propiedades de los sistemas, especificadas en términos de álgebra de procesos. La cuestión con este trabajo es que, si su objetivo era acercar la formalización a un público más amplio, logra exactamente lo contrario debido al alto contenido de lógica aplicada. No presenta un análisis previo desde el cual poder apalancar su propuesta.

Para Vilkomir et al. [14] las tareas importantes en la Ingeniería de Requisitos se orientan a resolver inconsistencias entre los reguladores y los desarrolladores de sistemas informáticos críticos. En su trabajo, proponen un nuevo enfoque para el proceso de regulación, incluyendo el uso de los requisitos reglamentarios formales como base para el desarrollo de métodos de evaluación de software. Se concentran en las diferencias entre las regulaciones y sugieren un enfoque intermedio. Argumentan que el paquete normativo debe incluir no solamente los requisitos reglamentarios, sino también los métodos para su evaluación. Este enfoque lo ilustran con ejemplos de requisitos para la protección de los sistemas informáticos de control, a través de accesos no autorizados y contra fallos comunes del software, usando la notación Z para la formalización. En su propuesta no tienen en cuenta el trabajo que otros han adelantado en este sentido y tampoco definen qué hacer en los sistemas no-críticos. Por eso es difícil determinar el desarrollo de la formalización de requisitos desde los aportes de su trabajo.

Morimoto et al. [15] proponen un método híbrido para verificar formalmente si la especificación de requisitos satisface los criterios de evaluación de seguridad, definidos en la norma ISO/IEC 15408. Clasifican los requisitos funcionales en estáticos y dinámicos, y formalizan los primeros con la notación Z y los segundos con lógica temporal. Como resultado, los desarrolladores pueden utilizar fácilmente el método para verificar si la especificación de un sistema satisface los requisitos funcionales de seguridad. De acuerdo con los autores, este método es una evolución y mejora los propuestos anteriormente, donde solamente se aplica la notación Z. Aunque es una propuesta válida para formalizar requisitos funcionales, no se encuentra referencias para un proceso similar con los no-funcionales. Además, no proponen ni analizan el estado de desarrollo de la formalización de requisitos, desde el cual pudieron extraer y estructurar su modelo.

Por su parte, Chatterjee y Johari [16] describen un enfoque simplificado y corroborativo para la formalización de requisitos. Su propuesta implica casos de uso, escenarios y diagramas de transición de estados, como base para la automatización del proceso de formalización, que logran a través de una herramienta de desarrollo propio, la cual ejemplifica el concepto subyacente e ilustra la facilidad de automatización. Previo a su propuesta hacen un análisis de las teorías, principios y aportes alrededor del desarrollo de la formalización de requisitos. El problema es que su estudio y propuesta se orientan al paradigma de la programación orientada por objetos, la cual, aunque de uso muy generalizado, todavía es un modelo estático. Esto contradice la necesidad del dinamismo de las soluciones a los complejos problemas actuales.

Para Serna [17] la aplicación de los métodos formales en la industria ha progresado ampliamente en la última década, y los resultados son prometedores. Describe ocho experiencias de casos reales, de las cuales cinco aplican la especificación formal de requisitos. A pesar de estos logros y de que se ha documentado en numerosos estudios, para este autor todavía es común el escepticismo acerca de su utilidad y aplicabilidad. Con los resultados obtenidos en estas experiencias encuentra un perfil de las necesidades de la industria, y propone que debe ser la comunidad de los métodos formales la encargada de darles solución. Concluye que los métodos formales son utilizados de forma generaliza y rodean todas las Ciencias Computacionales, pero todavía no tienen la acogida necesaria en los planes de estudios.

Peres y sus colegas [18] abordan una cuestión importante relacionada con la Ingeniería de Requisitos: *cómo guiar y ayudar la formalización de requisitos*. Con el fin de apoyar el proceso de formalización proponen una metodología basada en una estructura formal, como la piedra angular del proceso de refinamiento. Para estos autores los requisitos del sistema suelen ser escritos en un lenguaje natural, debido a que, por lo general, es del que los diferentes grupos de interés tienen un mayor entendimiento. Sin embargo, el uso de este lenguaje potencialmente da lugar a problemas de interpretación, que deben ser resueltos antes de usar técnicas de verificación automatizadas. Presentan la revisión a una serie de trabajos en los que se propone formalizar directamente los requisitos, que se dirigen esencialmente a la Ingeniería del Software, pero no describen cómo hacerlo en la Ingeniería de Requisitos.

2. MÉTODO

Este trabajo es el resultado de una revisión de la literatura en la que se aplica la metodología propuesta por Serna [19], con la cual se pretende encontrar respuesta a: 1) ¿qué se entiende por métodos formales? 2) ¿Qué se publica en la literatura en relación con la formalización de requisitos? y 3) ¿Qué tanto se ha progresado en la formalización de requisitos? Se conformó una muestra final de 40 trabajos relacionados, cuyos resultados sirvieron de base para encontrar las respuestas a estas preguntas.

3. RESULTADOS

3.1 Métodos Formales

Yan [20] afirma que los métodos formales son técnicas basadas en las matemáticas, a menudo con el apoyo de herramientas de razonamiento, que pueden ofrecer una manera rigurosa y eficaz para modelar, diseñar y analizar sistemas informáticos. Adicionalmente, [21-24] coinciden en que los métodos formales proporcionan una estructura matemática dentro de la cual, de manera sistemática, se utilizan para especificar, desarrollar y verificar un sistema. Para [23] los métodos formales permiten desarrollar un sistema informático de forma más completa, coherente e inequívoca, y para [25] evitan la introducción de imprecisiones o ambigüedades en el proceso.

Hay que tener en cuenta que la noción de *formal* en este contexto debe entenderse en términos generales como *lógica/matemática* [26]. Adicionalmente, Barlas et al. [27] manifiestan que estos principios matemáticos no necesitan conocimientos de lenguajes, y que pueden ser fácilmente entendidas en un contexto internacional. Al exponer algunos puntos de vista sobre los métodos formales, You et al. [28] reflexionan que desarrollar software correcto y confiable ha sido un problema continuo que debe resolverse con urgencia, y que una manera eficaz de lograrlo es la teoría formal, porque ofrece una posibilidad para crear software con pocos o ningún defecto [29]. Esto hace que un proceso así sea cualitativamente diferente a la Ingeniería del Software convencional, porque es fiable, robusto e ideal para utilizar en el diseño de sistemas de alta integridad que involucren cuestiones de seguridad, donde el costo de los errores puede ser muy alto [27].

El uso de métodos formales permite plantear de manera clara la especificación de un sistema, generando modelos que definen el comportamiento en términos de *qué debe hacer* y no de *cómo lo hace* [22]. Gracias a un correcto proceso de especificación se puede verificar propiedades derivadas de cada módulo mediante técnicas de razonamiento asociadas a los modelos formales, tales como probadores de teoremas y verificadores de modelos. Al recopilar opiniones de diferentes autores sobre métodos formales, se encontró que, generalmente, son costosos de aplicar [30], especialmente para la formación de los ingenieros; son lentos en su aplicación [26]; incrementan la calidad y la fiabilidad de un diseño y muestran su relación con los problemas prácticos y su potencial para el futuro; por otro lado, su aplicación exige que sea fácil de usar y que se tenga herramientas de verificación eficientes [21].

Además, los métodos formales pueden ayudar a resolver algunos de los problemas que enfrenta la industria del software actualmente, tales como la insatisfacción de los clientes por incapacidad para cumplir con los requisitos y altos costos de apoyo [30]. El objetivo principal de la Ingeniería del Software es permitirles a los desarrolladores crear sistemas que operen de forma fiable [23], y los métodos formales están destinados a proporcionar los medios para lograr una mayor precisión en el pensamiento y la documentación de esta ingeniería.

Las principales ventajas de los métodos formales son: las especificaciones formales son correctas y coherentes, y pueden ser probadas y verificadas; la ambigüedad en la especificación se evita automáticamente [23]; pueden garantizar la seguridad y la confiabilidad de los sistemas software [28]; se reduce en gran medida el error en el trabajo [28]; y se elimina los errores de diseño o inconsistencias antes de su implementación. Esto es esencial para el desarrollo de sistemas fiables, especialmente en aplicaciones de software crítico [31], los cuales presentan menos errores durante la verificación y, a menudo, requieren tiempos más cortos de desarrollo a la vez que consumen menos recursos económicos [32]. Estas ventajas se logran por el uso de las

matemáticas, porque son un medio para desarrollar un modelo exacto que puede describir el objeto y el funcionamiento de forma breve y precisa. Además, porque pueden representar unidades abstractas donde la especificación del sistema es coherente con las necesidades de clientes y usuarios. Por otro lado, las matemáticas ofrecen una manera de mostrar la contradicción y la imperfección en la especificación, a la vez que la coherencia entre el diseño y especificación [28].

En el otro extremo se encuentra las limitaciones de los métodos formales y, de acuerdo con [23], son difíciles de aprender y usar, y debido a la falta de modelos de desarrollo y herramientas los problemas complejos son difíciles de manejar. Para Mandrioli y Milano [32] nos encontramos con una actitud general *asocial* contra el razonamiento riguroso de los métodos formales. Lockhart, Purdy y Wilsey [31] concluyen que, desafortunadamente, hoy los métodos formales todavía son complejos y costosos para ser utilizados ampliamente en los sistemas software, porque el refinamiento de una especificación formal es un proceso complejo, lento y no está bien apoyado por herramientas.

Otra cuestión importante, pero negativa en el caso de los métodos formales, es que la mayoría de programadores los perciben como una teoría inútil, que no tiene ninguna relación con lo que hacen [32]. De acuerdo con estos investigadores no hay manera más rápida de perder la atención en una audiencia de programadores que mostrarles una fórmula matemática. Por desgracia, uno de los problemas centrales es que no ha habido avances comparables en métodos formales, y aunque surgen nuevos lenguajes y nuevas lógicas los errores en el diseño de programas, que provienen desde 1967, todavía se pueden encontrar en el software de hoy [33]. Serna [34] coincide en afirmar que, a pesar de sus ventajas significativas, los métodos formales no son ampliamente utilizados en el desarrollo de software industrial, y la academia todavía no demuestra el interés necesario para incluirlos en los planes de estudios.

3.2 La formalización de requisitos

Los requisitos son considerados la piedra fundamental sobre la que se puede desarrollar las soluciones software, y actualmente existe una creciente demanda de enfoques más rigurosos y sistemáticos para formalizarlos [35]. Serna y Serna [36] argumentan que los métodos formales se utilizan hoy para modelar sistemas de seguridad complejos y críticos, pero poco se trabaja en la formalización de los requisitos desde las primeras etapas de la Ingeniería de Requisitos.

La representación más utilizada para formalizar los requisitos son los lenguajes de especificación formal [35], porque los desarrolladores comprenden mejor el sistema y eluden las ambigüedades, los flujos, las omisiones y las inconsistencias. Además, la especificación es un importante mecanismo de comunicación entre los clientes y los diseñadores, entre los diseñadores y ejecutores, y entre los ejecutores y los probadores [37]. Pero las especificaciones formales no son documentos que se escriben una sola vez y, generalmente, no se realiza al principio del proceso de desarrollo de software.

Se necesita tiempo para crear una primera versión de utilidad que, luego de mucho esfuerzo y revisiones, permita desarrollar una especificación cercana a las necesidades que los clientes tienen en mente [20], y Bollin y Rauner [38] sostienen que un buen nivel de comprensión es una cuestión clave como atributo de calidad. Para Wolff [39], con el uso de un lenguaje de especificación formal el sistema puede ser descrito con precisión en cuanto a funcionalidad, concurrencia, integridad, exactitud, … Esto significa que las propiedades de un sistema se pueden analizar sin tener que ejecutarlo realmente.

Al utilizar la especificación formal, las propiedades del sistema se describen utilizando un lenguaje con sintaxis y semántica definidas matemáticamente [14]. Algunos lenguajes formales, tales como Z, B y VDM, se centran en especificar el comportamiento de los sistemas secuenciales donde los estados se describen en estructuras matemáticas, tales como conjuntos, relaciones y funciones [28]; mientras que métodos como CSP, CCS, Statecharts, Lógica Temporal y Autómatas, se centran en la especificación de los comportamientos del sistema en términos de secuencias, árboles u órdenes parciales de eventos [40]. El uso de lenguajes formales como Z y B puede mejorar la confianza del usuario en el sistema, y sus impactos en el uso del mismo [41]. Tamrakar y Sharma [42] afirman que Z, B y VDM son lenguajes de especificación formal utilizados para especificar las necesidades del usuario en un lenguaje matemático, cuyo producto puede ser probado y verificado automáticamente.

Z trabaja en alta abstracción a nivel del sistema y proporciona una sólida base para el diseño del mismo. Mediante el uso de este lenguaje, no solamente se descubre más errores en la especificación, sino también en las fases de pruebas y mantenimiento. Esto es conveniente, porque con la ingeniería tradicional el costo de corregir errores en las fases posteriores es mucho mayor que en las primeras [41]; además, Z es una manera de descomponer una especificación en pequeñas partes llamadas esquemas. Por su parte, B es uno de los métodos formales más conocidos. Se basa en la lógica de primer orden, la teoría de conjuntos, la aritmética de enteros y las sustituciones generalizadas; se utiliza para el diseño de software desde los requisitos funcionales, y permite producir casos de prueba que demuestran la exactitud y la consistencia del modelo software aplicado. El objetivo de B es obtener un producto probado y fiable [43]; mientras que VDM se utiliza para demostrar la equivalencia de los conceptos del lenguaje [42].

De acuerdo con Pandey y Batra [35], en la actual era digital las empresas se enfrentan al desafío de liberar proyectos software de calidad, a tiempo y dentro del presupuesto, pero la realidad es que lo entregan con errores, con poca funcionalidad y a veces con sobrecostos. El costo extra se genera debido a errores en la especificación de requisitos, que puede costar mucho tiempo y dinero corregirlos, cuando se detectan en las fases tardías del ciclo de vida. Por lo tanto, en la especificación de requisitos se debe seleccionar una metodología formal para abordar la fiabilidad durante la Ingeniería de Requisitos y el diseño, porque los métodos formales permiten desarrollar los errores antes de la liberación del producto [31].

Los errores de especificación pueden ser reducidos drásticamente mediante el uso de métodos formales y, en consecuencia, el ingeniero de software puede crear una especificación más completa, coherente e inequívoca que con los métodos convencionales [23, 41]. Por su parte, Bollin y Rauner [38] manifiestan que una buena especificación formal es sintáctica y semánticamente correcta, que permite un mapeo sin pérdidas entre todos los conceptos de la especificación y el modelo mental del sistema especificado; también agregan que debe ser completa, coherente y adecuada, y tener en cuenta que la facilidad de comprensión es un requisito esencial para decidir sobre su corrección semántica.

No se encuentra una herramienta informática que pueda garantizar la corrección completa de un modelo de computador [42], por lo tanto, aunque la especificación esté escrita utilizando cualquiera de los lenguajes formales, siempre contiene un alto potencial de errores. Significa que el arte de escribir una especificación formal no asegura que el sistema desarrollado sea coherente, correcto y completo. Por otro lado, si la especificación se comprueba y se analiza con el apoyo de herramientas automatizadas, aumenta la confianza sobre el sistema con la posible identificación de los errores potenciales, si existen, en la sintaxis y la semántica de la descripción formal. Una de las ideas más difundidas sobre el uso de métodos formales es su casi nula aplicación en la

industria software. Si bien su adopción ha sido lenta, existe casos de importantes empresas que han tenido éxito al aplicarlos en proyectos reales [22].

Para conocer qué se publica en la literatura en relación con la formalización de requisitos, en la Tabla 1 se presenta el resumen de los resultados de esta investigación. La línea de tiempo de observación es a partir de 2010. Aunque es conocido que la mayor parte del trabajo en esta área se presentó en la segunda mitad del siglo pasado, el objetivo de esta investigación era verificar su progreso a principios del siglo XXI. Los trabajos de la muestra final se clasificaron de acuerdo con el enfoque primario presentado. Se aclara que muchos autores presentan resultados en más de una tipología, es decir, su investigación puede ser teórica y al mismo tiempo experimental.

Tabla 1. Qué se publica en la literatura acerca de la formalización de requisitos

Medio		Teórica	Experimental	Práctica	Industrial
Revista	66%	63%	34%	20%	23%
Conferencia	34%				

- *Investigación teórica*: artículos que contienen definiciones o descripciones acerca de la formalización de requisitos.
- *Investigación experimental*: demuestra resultados a partir de experimentos en laboratorio.
- *Aplicación práctica*: describen y aplican métodos, técnicas o procedimientos de formalización de requisitos en casos de estudio.
- *Aplicación industrial*: en los que la formalización de requisitos se aplica en casos reales.

3.3 Progreso del trabajo en la formalización de requisitos

En la Tabla 2 se presenta la tendencia del progreso de la formalización de requisitos, de acuerdo con los resultados de esta investigación.

Tabla 2. Progreso de la formalización de requisitos

Fuente	Aporte
[44]	Presenta un uso de los métodos formales en la formalización de requisitos centrados en el usuario. Los resultados son prometedores en cuanto al mejoramiento de la calidad del producto software.
[43]	Analiza el método B para la especificación de requisitos y describe los casos de uso y las propiedades de seguridad. Aunque su aplicación es sencilla, es un aporte hacia la masificación de los métodos formales en la especificación de requisitos.
[45]	Propone una metodología para formalizar la Ingeniería de Requisitos y el diseño, orientada a analizar y gestionar requisitos. Esta investigación brinda luces para demostrar la utilidad de los métodos formales en la Ingeniería de Requisitos.
[46]	Plantea el análisis y el diseño formal de la seguridad de los requisitos utilizando especificaciones en B. Los resultados demuestran efectividad y capacidad en la prpducción de software seguro.
[47]	Describe los métodos formales más comunes y la importancia de aplicar la especificación formal en los primeros pasos del ciclo de vida. Enfrenta las posiciones de los críticos y defensores de los métodos formales y presenta ejemplos de historias de éxito en la industria.
[48]	Utiliza la notación Z para formalizar los requisitos, buscando protección y seguridad contra amenazas internas en un caso de estudio de éxito.
[49]	Desarrolla una metodología y técnicas para formalizar y validar requisitos en aplicaciones críticas de seguridad. Aplican el resultado en un caso de estudio con resultados mejorados.
[34]	Describe la aplicación de la especificación formal de requisitos y el mejoramiento que se logra en la calidad del producto final. Propone la incoporación de los métodos formles en los contenidos curriculares de los programas en Ciencias Computacionales.
[50]	Presenta un método práctico para usar especificaciones formales en pruebas de registro en Mars Science Laboratory. Demuestra las ventajas de utilizar un lenguaje de especificación formal.

[51]	Integra los conceptos de los métodos formales desde el inicio de la programación y, mediante una didáctica simple, muestra cómo hacer una especificación formal y qué modelos se pueden utilizar.
[52]	Es una revisión a la aplicación de los métodos formales en proyectos reales en la industria. Presenta los diversos casos de éxito y describe la situación actual en cuanto a la aplicación de los métodos formales.
[53]	Hace un recorrido histótico por los métodos formales y describe sus aplicaciones y beneficios en la formalización de requisitos. Propone que la investigación en esta área se debe orientar a aplicaciones prácticas y a buscar reducir los costos de su utilización.
[54]	Aplica una base formal para la especificación y verificación de requisitos, presenta una definición formal y una teoría de composición formal para la Ingeniería de Requisitos. Aplica la verificación formal y describe los resultados.
[20]	Describe la aplicación de la especificación formal en Communications-Based Train Control (CBTC), obteniendo una descripción profunda y exacta del sistema, y lenguaje Z en el desarrollo. Concluye que al usar métodos formales se gana confianza para asegurar el sistema.
[55]	Presenta un nuevo enfoque para la especificación y verificación formal basado en la reescritura la lógica. Luego lo aplica en caso de estudio con resultados prometedores en mejoramiento de la calidad.
[23]	Estudio comparativo entre los métodos de especificación formal y descripción de las razones de por qué usarlos en los procesos industriales.
[28]	Muestra los logros y problemas de los métodos formales y valida su aplicación para garantizar la seguridad y estabilidad del sistema.
[27]	Presenta una comparación entre la especificación de requisitos tradicional y la formal. Con la segunda logra mayor comprensión, elimina ambigüedades y obliga a una mejor precisión de la especificación. Muestra aplicaciones y beneficios de su uso en la industria.
[39]	Utiliza la experiencia en la industria para combinar el desarrollo ágil Scrum con los métodos formales, pero desde una conceptualización teórica. Presenta una evaluacion y discusión acerca de los beneficios de utilizarlos para formalizar los requisitos.
[56]	Desde un marco teórico concluye que el proceso de la formalización de los requisitos es un problema de adaptación transcultural, analiza los pros y los contras del mismo y da a conocer un modelo refinado para un proceso de desarrollo de software formal.
[57]	Investigación teórica en la que se conlcuye se conlcuye que los métodos formales ha alcanzado significativos avances, lo que vislumbra aplicaciones ambiciosas en el futuro para apoyar rigurosos y coherentes planes de estudios en Ciencias Computacionales.
[35]	Discute los diversos lenguajes de especificación formal y analiza sus fortalezas y debilidades. Realiza recomendaciones para el uso de métodos formales, especialmente en la Ingenierpia de Requisitos.
[58]	Mediante una serie de recomendaciones para formalizar los requisitos, aporta al logro práctico de la selección de una tecnica específica. Su objetivo es mejorar la aceptación de los métodos formales en el modelado de requisitos.
[59]	La especificación formal todavía tiene usos limitados, pero la comunidad tiene una comprensión diferente acerca de su utilidad y necesidad. Hasta el momento, su desarrollo se focaliza principalmente en evaluar las herramientas relacionadas.
[60]	Analiza el dominio de los métodos formales para identificar defectos y disminuir fracasos. Aplica la formalización en la especificación, el modelado y la verificación.
[61]	Propone una metodología de modelado de requisitos para convertir los informales en formales. Con ellos busca eliminar ambigüedades y defectos, refinar y perfeccionar el sistema y servir como un medio de comunicación entre las partes interesadas.
[62]	Un estudio de los lenguajes de especificación formal, describe los modelos que se pueden aplicar y realiza un análisis general. Realiza una experimentación con estadística de depuración en un estudio de caso.
[63]	Aplica la especificación formal por medio de Z para potencializar la seguridad del sistema y reducir las amenazas. Los resultados son prometedores para ampliaciones y elaboraciones futuras.
[31]	Demuestra la utilidad de los métodos formales para la especificación de requisitos en sistemas de seguridad críticos. La especificación formal reduce la ambigüedad del diseño, ayuda a probar la consistencia, e incrementa la confianza y el rendimiento.
[64]	Propone reglas de transformación desde el modelo B y describe los requisitos con máquinas abstractas. Para el autor, la especificación formal es un proceso intermedio entre la especificación de requisitos y la escritura del código.
[65]	Realiza una revisión de la literatura, hace un recorrido por la esencia, función, uso e inconvenientes de las técnicas de especificación formal y analiza criterios de valoración y evaluación a sus debilidades. Busca que la especificación formal se afiance como actividad básica de investigación y formación en la academia.
[38]	Muestra una serie de recomendaciones para realizar especificaciones formales, e identifica que la facilidad de lectura de una especificación es un factor clave para mejorar la calidad del software.

[40]	Aplica gramática formal con anotaciones semánticas para formalizar requisitos, de tal forma que puedan ser analizados y procesados por el computador. Los requisitos formulados pueden ser analizados y procesados en las primeras etapas de desarrollo.
[66]	Afirma que muchos de los problemas desafiantes en la construcción de sistemas requieren el apoyo formal para su modelado y análisis. En los procesos de investigación y aplicación de las Ciencias Computacionales existe un número creciente de aplicaciones. Pero todavía no constituyen una parte integral de los procesos formativos en pregrado y posgrado.
[41]	Utiliza especificación formal Z para mejorar la calidad y confianza del sistema. Muestra trabajos relacionados y aplica una propuesta en un estudio de caso en salud.
[67]	Aplica object-Z para describir sistemas complejos. Demuestra su utilidad en un caso de estudio de un sistema de suministro de gasolina. Los resultados obtenidos alientan el uso de la especificación formal.
[68]	Describe aspectos relevantes de los métodos formales y realiza un marco del futuro de la investigación en el área. Los resultados invitan a la reflexión de este componente de las Ciencias Computacionales y de la necesidad de tener una comunidad más amplia y sólida dedicada a fomentar y aplicar la formalización de los requisitos.
[29]	Realiza un estudio empírico para encontrar la productividad de los proyectos que utilizan métodos formales, específicamente al formalizar los requisitos. Identifica una serie de preguntas sobre los resultados en productividad de estos proyectos.
[42]	Compara tres métodos para especificar formalmente: Z, B y VDM. Usar especificaciones formales no asegura un sistema completamente correcto, pero se aumenta la confianza en el mismo.
[69]	Utiliza el método Event-B para escribir especificaciones formales, con el fin de garantizar la compresión de los límites en los que un algoritmo puede ser usado. Para el autor, la especificación, la validación y la verificación formales con la clave para obtener un mejor diseño.
[70]	Realiza una representación de formalización de requisitos por medio de diagramas SysML usando la semántica RSL. Formaliza únicamente los requisitos no-funcionales y ese diseño lo aplica a un caso de estudio en el área automotriz.

4. ANÁLISIS DE RESULTADOS

De acuerdo a la revisión de la literatura en esta investigación, el trabajo en formalización de requisitos no ha tenido el auge que se esperaba en el siglo XXI. Algunos autores analizan este hecho y presentan sus conclusiones. Entre las causas se destacan: falta de formación en la academia, falta de aceptación en la industria, no hay demostraciones realmente importantes de sus ventajas, los costos son altos, falta de personal capacitado, los ingenieros les tienen fobia a las matemáticas, la industria no quiere experimentar con principios que no tienen comprobación suficiente y la investigación en métodos formales ha perdido interés y patrocinio.

Los investigadores de mediados del siglo pasado estaban convencidos que la única manera de mejorar la calidad de los productos software era a través de la matematización de la Ingeniería del Software, pero a medida que sus resultados comenzaron a publicarse la industria comenzó a encontrar y colocar barreras al progreso de este trabajo. Solamente el software crítico los arropó como tabla de salvación para solucionar sus problemas de fiabilidad y seguridad, y son diversos los casos de éxito en estos desarrollos.

El problema surge cuando las empresas de desarrollo de software comercial se enfrentan a situaciones de incumplimiento, que incrementan los costos del proceso y que las obliga a dedicar menos tiempo a la estructura misma de las actividades. La decisión es disminuir la mayor cantidad de tiempo posible, y como la forma tradicional de realizar la verificación y validación es como una barrera de contención al final del ciclo de vida, esta fase es la que se recorta para encontrar algo de tiempo.
Además, el progreso de la formalización de requisitos es lento, porque la academia no capacita en métodos formales. Este tema se ha relegado a unos pocos investigadores que convencen a sus estudiantes de posgrado para que trabajen con ellos. Con el inconveniente de que los profesionales le temen a todo lo que tenga que ver con matemáticas. La recomendación es a que

las universidades incluyan cursos relacionados en los planes de estudios de los pregrados en Ciencias Computacionales, de tal forma que los estudiantes se interesen y profundicen en su asimilación y comprensión.

De acuerdo con los autores analizados en esta investigación, la formalización de requisitos se investiga de forma mayoritariamente teórica, y de esta manera no se logrará convencer a la industria de su aplicación. Puede ser que la etapa de la teorización ya haya sido superada en los muchos trabajos que se presentaron el siglo pasado. La necesidad actual es masificar los métodos formales y encontrar la manera de aplicarlos en la matematización de la Ingeniería del Software, pero con costos que estén al alcance de las empresas desarrolladoras de software.

De esta manera será posible reavivar el interés en esta área de trabajo y de encontrar el patrocinio necesario para ejecutar la investigación necesaria. Si el objetivo es lograr por fin superar la crisis del software, los métodos formales deben ser el centro de desarrollo, porque sus bondades ya han sido suficientemente demostradas y porque el lenguaje con el que funciona el computador es matemático.

5. CONCLUSIONES

En este capítulo se presenta el desarrollo y el progreso de la formalización de requisitos, como resultado de una investigación basada en una revisión de la literatura. El objetivo fue determinar el estado actual de la investigación en esta área, con el ánimo de presentarle a la comunidad un reporte del desarrollo que ha alcanzado en el siglo XXI.

Los resultados demuestran que su progreso y desarrollo son lentos, y que en los últimos años el interés por darle continuidad ha disminuido. Aunque se acepta que los métodos formales son una herramienta útil y necesaria para superar la crisis de la calidad del software, muy pocos investigadores quieren incrementar su nivel de madurez o de aceptación. La academia tiene parte de culpa en esta situación, porque todavía no los incluye en los planes de estudios de los pregrados, y algunas los relegan solamente a algunos posgrados.

La industria necesita demostraciones de los beneficios de la formalización, pero hasta ahora no tiene inclinación directa por apoyar las iniciativas que en este sentido propone la comunidad.

Solamente en el desarrollo de sistemas críticos se aprecia una amplia participación de las empresas de desarrollo y la industria, porque el objetivo es que funcionen sin poner en riesgo la vida humana o las inversiones económicas. Se necesita mayor trabajo para reducir los costos actuales de la formalización, pero, sobre todo, es urgente contar con el personal capacitado para aplicarla y darle continuidad.

Somos sociedad software-dependiente, porque muy pocas de sus actividades están por fuera del ámbito de este desarrollo tecnológico. Pero los productos que se liberan o entregan a los usuarios todavía no satisfacen aspectos como fiabilidad y seguridad. Las matemáticas ofrecen la posibilidad de superar este problema y su representación en los métodos formales es una alternativa prometedora.

REFERENCIAS

[1] Chalupnik M. et al. (2009). Approaches to mitigate the impact of uncertainty in development processes. En 17th International Conference on Engineering Design. Stanford, USA.

[2] Dahlstedt A. y Persson A. (2003). Requirements interdependencies - Molding the state of research into a research agenda. En Ninth International Workshop on Requirements Engineering. Klagenfurt/Velden, Austria.
[3] Heimdahl M. y Leveson N. (1995). Completeness and consistency analysis of state-based requirements. IEEE Transactions on Software Engineering 22(6), 3-14.
[4] Heitmeyer C. et al. (1996). Automated consistency checking of requirements specifications. ACM Transactions Software Engineering and Methodology 5(3), 231-261.
[5] Yu L. et al. (2008). Completeness and consistency analysis on requirements of distributed event-driven systems. En 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering. Washington, USA.
[6] Post A. et al. (2011). Vacuous of real-time requirements. En 19th IEEE International Requirements Engineering Conference. Trento, Italy.
[7] Post A. et al. (2011). Rt-inconsistency: A new property for real-time requirements. Lecture Notes in Computer Science 6603, 34–49.
[8] Post A. et al. (2011). Podelski. Applying restricted English grammar on automotive requirements - Does it work? A case study. Lecture Notes in Computer Science 6606, 166–180.
[9] Jureta I. et al. (2008). Schobbens. Clear justification of modeling decisions for goal-oriented requirements engineering. Requirements Engineering 13, 87–115.
[10] Boehm, B. (1984). Model and metrics for software management and engineering. IEEE.
[11] Boehm, B. (1987). Industrial software metrics top 10 list. IEEE Software 4(5), 84–85.
[12] Mathiassen L. y Munk A. (1985). Formalization in systems development. Lecture Notes in Computer Science 186, 101-116.
[13] Fantechi A. et al. Assisting requirement formalization by means of natural language translation. Formal Methods in System Design 4(3), 243-263.
[14] Vilkomir S. et al. (2006). Formalization and assessment of regulatory requirements for safety-critical software. Innovations in Systems and Software Engineering 2(3), 165–178.
[15] Morimoto S. et al. (2008). Classification, formalization and verification of security functional requirements. Lecture Notes in Computer Science 4910, 622-633.
[16] Chatterjee R. y Johari K. (2010). A simplified and corroborative approach towards formalization of requirements. Communications in Computer and Information Science 94, 486-496.
[17] Serna A. (2012). Formal Methods in industry. Los Métodos Formales en la Industria. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 2(2), 44-51.
[18] Peres F. et al. (2012). A formal framework for the formalization of informal requirements. The International Journal of Soft Computing and Software Engineering 2(8), 14-27.
[19] Serna E. (2018). Metodología de investigación aplicada. En E. Serna (Ed.), Ingeniería: Realidad de una Disciplina (pp. 6-33). Editorial Instituto Antioqueño de Investigación.
[20] Yan F. (2011). Studying formal methods applications in CBTC. En International Conference on Management and Service Science. Wuhan, China.
[21] Seceleanu C. (2011). Formal methods applied in industry: Success stories, limitations, perspectives – panel introduction. En IEEE 35th Annual Computer Software and Applications Conference. Munich, Germany.
[22] Ambrosio M. y Andrade G. (2011). Métodos formales aplicados en la industria del software. Temas de Ciencia y Tecnología 15(43), 3-12.
[23] Kaur A. et al. (2012). A comparative study of two formal specification languages: Z-notation & B-method. En Second International Conference on Computational Science, Engineering and Information Technology. Coimbatore, India.
[24] Barbosa L. y Lumpe M. (2014). Formal aspects of component software. Lecture Notes in Computer Science 7684, 253–254.
[25] González C. y Cabot J. (2014). Formal verification of static software models in MDE: A systematic review. Information and Software Technology 56(8), 821–838.
[26] Gruner S. (2011). Editorial: Special section on formal plus agile methods. ACM Software Engineering Notes 36(4), 26.
[27] Barlas K. et al. (2012). Extending standards with formal methods: Open document architecture. En International Symposium on Innovations in Intelligent Systems and Applications. Trabzon, Turkish.

[28] You J. et al. (2012). A survey on formal methods using in software development. En International Conference on Information Science and Control Engineering. Shenzhen, China.
[29] Jeffery R. et al. (2015). An empirical research agenda for understanding formal methods productivity. Information and Software Technology 60, 102–112.
[30] Shirali S. y Shirali M. (2010). Using formal methods in component based software development. En T. Sobh (Ed.), Innovations and advances in computer sciences and engineering (pp. 429-432). Springer.
[31] Lockhart J. et al. (2013). Formal methods for safety critical system specification. En 57th International Midwest Symposium on Circuits and Systems. College Station, USA.
[32] Mandrioli D. y Milano P. (2015). On the heroism of really pursuing formal methods - Title inspired by Dijkstra's 'On the Cruelty of really Teaching Computing Science'. En Third FME Workshop on Formal Methods in Software Engineering. Austin, USA.
[33] Parnas D. (2010). Really rethinking 'formal methods'. Computer 43(1), 28–34.
[34] Serna E. (2010). Formal methods and Software Engineering. Revista Virtual U. Cató. del N. 30, 158-184.
[35] Pandey S. y Batra M. (2013). Formal methods in requirements phase of SDLC. International Journal of Computer Applications 70(13), 7–14.
[36] Serna E. y Serna A. (2014). Formal specification in context: Current and future. Ingeniare - Revista Chilena de Ingeniería 22(2), 243-256.
[37] Krad, H. (2011). Formal methods and automation for system verification. En 4th International Conference on Modeling, Simulation and Applied Optimization. Kuala Lumpur, Malaysia.
[38] Bollin A. y Rauner D. (2014). Formal specification comprehension the art of reading and writing Z. En 2nd FME Workshop on Formal Methods in Software Engineering. Hyderabad, India.
[39] Wolff S. (2012). Scrum goes formal : Agile methods for safety-critical systems. En First Intern.ational Workshop on Formal Methods in SE: Rigorous and Agile Approaches. Zurich, Switzerland.
[40] Schraps M. y Peters M. (2014). Semantic annotation of a formal grammar by Semantic Patterns. En 4th International Workshop on Requirements Patterns. Karlskrona, Sweeden.
[41] Azeem M. et al. (2014). Specification of e-Health System using Z: A motivation to formal methods. En International Conference for Convergence of Technology. Pune, India.
[42] Tamrakar S. y Sharma A. (2015). Comparative study and performance evalua-tion of formal specification language based on Z, B and VDM tools. Inter. J. of Scie. & Eng. Research 6(9), 1540–1543.
[43] de Sousa T. et al. (2010). Automatic analysis of requirements consistency with the B method. Software Engineering Notes 35(2), 1-4.
[44] Bhavsar M. (2010). Analysis of Multiagent Based Interactive Grid Using Formal Methods - A Reliable Approach. En 3rd Intern. Conference on Emerging Trends in Engineering and Technology. Goa, India.
[45] Honghao G. (2010). Based on formal methods in trustable software requirements engineering. En International Conference on Internet Technology and Applications. Wuhan, China.
[46] Hassan R. et al. (2010). Formal analysis and design for engineering security automated derivation of formal software security specifications from goal-oriented security requirements. IET Software 4(2), 149–160.
[47] van Der Poll J. (2010). Formal methods in software development: A road less travelled. South African Computer Journal 45, 40-52.
[48] Singh M. (2010). Formal Specification of Common Criteria Based Access Control Policy Model International Journal of Network Security 11(3), 139-148.
[49] Cimatti A. et al. (2010). Formalization and Validation of Safety-Critical Requirements. En Workshop on Formal Methods for Aerospace. Eindhoven, Netherlands.
[50] Barringer H. et al. (2011). An entry point for formal methods: Specification and analysis of event logs. En Workshop on Formal Methods for Aerospace. New Mexico, USA.
[51] Bishop M. et al. (2011). Applying formal methods informally. En Annual Hawaii International Conference on System Sciences. Kauai, USA.
[52] Fernández C. et al. (2011). Métodos formales aplicados en la industria del software. Temas de Ciencia y Tecnología 15(43), 3-12.
[53] Serna E. (2011). Formal Methods: Perspective and future application. En III Jornadas de Investigación de la Facultad de Ingenierías. Medellín, Colombia.
[54] Ibrahim N. et al. (2011). Specification and Verification of Context-dependent. En 20th international conference on World Wide Web. Hyderabad, India.

[55] Ammar B. y Abdallah K. (2011). Towards the formal specification and verification of multi-agent based systems. International Journal of Computer Science Issues 8(4), 200–210.
[56] Bollin A. (2013). Do you speak Z? Formal methods under the perspective of a cross-cultural adaptation problem. En 1st FME Workshop on Formal Methods in Software Engineering. San Francisco, USA.
[57] Serna E. y Serna A. (2013). Challenges and opportunities of research in formal methods. En XII Conferencia Iberoamericana en Sistemas, Cibernética e Informática. Orlando, USA.
[58] Atlee J. et al. (2013). Recommendations for Improving the usability of formal methods for product lines En 1st FME Workshop on Formal Methods in Software Engineering. San Francisco, USA.
[59] Serna E. y Serna A. (2013). Formal Specification - Present and Future. En Informática 2013. La Habana, Cuba.
[60] Chen X. (2013). Research on the implementation of internal control in enterprise information system by domain analysis and formal methods - A case study of sales activities internal control under Chinese enterprise environment. En Fourth World Congress on Software Engineering. Hong Kong, China.
[61] Chan L. et al. (2013). Rule-based behavior engineering: Integrated, intuitive formal rule modelling. En 22nd Australian Software Engineering Conference. Melbourne, Australia.
[62] Diallo R. (2013). The need for usable formal methods in verification and validation. En Winter Simulation Conference. Washington, USA.
[63] Noaman M. et al. (2013). The specifications of E-Commerce secure system using Z language. The Research Bulletin of Jordan ACM II(III), 127-131.
[64] Wu T. et al. (2013). Formal specification and transformation method of system requirements from B Method to AADL Model. En 17th International Conference on Computational Science and Engineering. Chengdu, China.
[65] Serna E. y Serna A. (2014). Formal specification in context: Current and future. Ingeniare - Revista Chilena de Ingeniería 22(2), 243-256.
[66] Serna E. y Serna A. (2014b). Formal methods in context. En XIII Conferencia Iberoamericana en Sistemas, Cibernética e Informática. Orlando, USA.
[67] Li Y et al. (2014). Specifying complex systems in Object-Z: A case study of petrol supply systems. Journal of Software 9(7), 1707–1717.
[68] Serna E. y Serna A. (2014c). Perspective and application of the formal methods. En XIII Conferencia Iberoamericana en Sistemas, Cibernética e Informática. Orlando, USA.
[69] Singh A. y Yadav D. (2015). Formal specification and verification of total order broadcast through destination agreement using e Vent-B. International Journal of Computer Science & Information Technology 7(5), 85–95.
[70] Walter S. (2015). Towards formalized model-based requirements for a seamless design approach in safety-critical systems development. En 18th International Symposium on Real-Time Distributed Computing Workshops. Auckland, USA.

CAPÍTULO X
La especificación formal en contexto: Actual y futuro[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

La especificación formal es un área de investigación en la Ingeniería del Software de este siglo, en la que se aplica en diversas configuraciones y técnicas, y aunque su uso industrial todavía es limitado, la comunidad científica tiene actualmente una comprensión diferente acerca de su utilidad y necesidad. Hasta el momento el trabajo de los investigadores se focaliza en la especificación escrita durante el diseño del modelo funcional preliminar, por lo que se centra principalmente en evaluar las herramientas relacionadas. En este trabajo se realiza una revisión a la literatura, se hace un recorrido por la esencia, la función, el uso y los inconvenientes de las técnicas de especificación formal, y se analiza algunos criterios de valoración y de evaluación a sus debilidades. Los resultados se convierten en la base para formular trabajos futuros, con el objetivo de buscar que la especificación formal se afiance como actividad básica de investigación.

[1] Publicado en Ingeniare. Revista chilena de ingeniería 22(2), 243-256. 2014.

INTRODUCCIÓN

Desde el surgimiento de las Ciencias Computacionales los investigadores han considerado a la especificación formal como una de sus áreas de interés. Finalizando los años cuarenta Turing [1] observó que el razonamiento acerca de programas secuenciales era más sencillo cuando, en puntos específicos del mismo, se hacía anotaciones acerca de las propiedades de su estado. En los años sesenta Floyd [2], Hoare [3] y Naur [4] propusieron técnicas axiomáticas para demostrar la consistencia entre los programas secuenciales y esas propiedades, a las que llamaron *especificaciones*, y Dijkstra [5] demostró cómo utilizar constructivamente el cálculo para derivar programas no-determinísticos que las cumplieran.

Parnas [6] y Liskov [7] propusieron técnicas específicas para expresar formalmente las propiedades de los programas, particularmente para datos estructurados, y Pnueli [8] lo hizo para programas concurrentes. Estos aportes conformaron el punto de partida para la *especificación formal* como un área de investigación [9, 11], y desde entonces se ha incrementado continuamente el interés que despertó, lo mismo que los múltiples usos en la Ingeniería del Software [12-14]. Hasta el momento el objetivo principal de esta comunidad de investigadores se focaliza en la especificación escrita durante el diseño del modelo funcional preliminar [12], por lo que lo que su investigación, alrededor de la cual se genera este trabajo, se centra principalmente en evaluar y comparar las herramientas relacionadas con esta especificación y en analizar sus fortalezas y debilidades.

En este capítulo se presenta el resultado de una revisión a la literatura para determinar la esencia, la función, el uso y los inconvenientes de las técnicas de especificación formal. Se discute algunos criterios de valoración y se hace una evaluación a sus fortalezas y debilidades. Los resultados se convierten en la base para formular una serie de trabajos futuros, con el objetivo de que la especificación formal se afiance como una actividad básica de investigación en la ingeniería de software de este siglo.

1. MÉTODO

El procedimiento para realizar esta investigación se orientó a identificar estudios y opiniones, que pudieran ser candidatos a incluir o excluir del conjunto final de la revisión. Las fuentes fueron los buscadores en internet, las bases de datos digitales de periódicos, revistas, asociaciones industriales y académicas, y en las bibliotecas. Los parámetros de búsqueda incluyeron palabras como *formal specification* y *formal languages*, combinadas con *techniques*, *metodologies*, *methods*, *research*, *industrial application*, *requirments engineering* y *software engineering*. Estas combinaciones debían aparecer en el título o en el contenido del documento. Para lograr mayor cubrimiento no se determinó una línea de tiempo específica y no se excluyó ninguna fuente inicial. Además, en los estudios primarios seleccionados se validó la solidez de la metodología aplicada, y los resultados presentados y las opiniones y/o análisis las debían presentar instituciones o personas que estuvieran relacionadas con el área de investigación, y que su trayectoria fortaleciera el postulado.

En la muestra final se incluyeron directamente: 1) los artículos de revistas en bases de datos, por haber pasado por procesos de selección y evaluación estructurados, 2) los documentos publicados en los *blogs*, cuyos autores y empresas fueran idóneos en el área, y 3) los artículos en periódicos y revistas, los cuales debían haber sido escritos por autores con trabajo en especificación formal y experiencia en los campos relacionados. Al final la muestra inicial quedó conformada por 154 trabajos. Además de estas características debían hacer, en su totalidad, un

aporte relevante a la temática de investigación. Para alcanzar los objetivos se aplicaron las siguientes fases:

1. Identificar los estudios relevantes.
2. Excluir estudios con base en el título.
3. Excluir estudios con base en los resúmenes.
4. Analizar los estudios y seleccionar los más relevantes para la temática con base en el texto.

Los criterios de inclusión y exclusión más importantes fueron: formalidad y pertinencia del sitio donde se aloja, autoridad del o los autores, calidad y aportes del contenido, fuentes de datos, sustentación de la tesis, calidad de la investigación y coherencia de los resultados. Luego de este proceso se excluyeron 38 trabajos de la muestra inicial, y con los 116 restantes se realizó el análisis que se presenta a continuación.

2. RESULTADOS

2.1 Especificación formal

El término Especificación Formal se puede referir a diversos aspectos en el ciclo de vida del software y se utiliza indistintamente para un producto como para su correspondiente proceso, por lo que está sobrecargado en la literatura. Generalmente, se acepta que una especificación formal es la expresión, en algún lenguaje formal y con cierto nivel de abstracción, de una serie de características que el sistema debe satisfacer. Esta definición se utiliza dependiendo de lo que se comprenda como *sistema*, del tipo de *características* a satisfacer, del nivel de *abstracción* aplicado y del *lenguaje formal* utilizado [15]. El sistema puede ser un modelo descriptivo del dominio de las necesidades del software y su entorno, del software como tal, de la interfaz de usuario, de la arquitectura del software o de algún proceso a seguir, entre otros. Las características se pueden referir a los objetivos de alto nivel, a los requisitos no-funcionales, a la hipótesis del dominio, o a los protocolos de interacción entre esos componentes.

Pero para asegurar que una solución resuelve correctamente un problema, y más allá de las diferentes concepciones de especificación, existe una idea común relacionada con el dominio del problema: *se debe establecer el primer estado en el que éste es correcto*. Sin embargo, esta dicotomía es simplista, porque generalmente una solución se puede representar como un conjunto de sub-problemas, los cuales se especifican y resuelven a la vez [16]; por lo tanto, una especificación debe cumplir con alguna característica de alto nivel y también satisfacer algunas de bajo nivel.

Otra cuestión que se determinó en esta investigación es que, frecuentemente, el término *formal* se confunde con *precisión*, y aunque la incluye, lo contrario no es cierto. El lenguaje para representar una especificación formal se expresa con: 1) reglas de *sintaxis*, para determinar oraciones gramaticalmente bien formadas, 2) reglas *semánticas*, para interpretar oraciones de forma precisa y significativa en el dominio, y 3) reglas de *inferencia* para deducir información útil desde la especificación (teoría de las pruebas), que constituyen la base para automatizar el análisis. Normalmente, el conjunto de características que se especifica es grande, por lo que el lenguaje debe permitir que se organicen en unidades vinculadas a través de estructuras de relación, como la especialización, la agregación, la instanciación, el enriquecimiento, el uso, entre otras. Cada una de estas unidades tendrá: 1) una parte declarativa, donde se declara las variables de interés, y 2) una parte de aserción, donde se formalizan las propiedades del objeto en las variables declaradas.

Por otro lado, escribir una especificación *correcta* es difícil, probablemente tanto como escribir un programa *correcto.* Se puede considerar *buena* si cumple con: 1) ser *adecuada*, indicar adecuadamente el estado del problema en cuestión, 2) tener *consistencia interna,* poseer una interpretación semántica significativa que haga verdadero a todo el conjunto de propiedades especificadas, 3) no ser *ambigua*, tener una sola interpretación de lo que es cierto, 4) ser *completa*, con respecto al nivel superior donde las propiedades especificadas sean suficientes para establecerla [17], 5) ser *satisfecha*, por el nivel inferior, y 6) ser *mínima*, no puede tener propiedades de estado irrelevantes para el problema o que solo lo sean para la solución del mismo [18].

2.1.1 Ventajas de la formalización

La formalización es un proceso necesario para diseñar, validar, documentar, comunicar, hacer re-ingeniería y reutilizar soluciones informáticas. Además, permite obtener especificaciones con mayor nivel de calidad y proporciona las bases para su automatización. Otras aplicaciones la describen para plantear preguntas y para detectar problemas serios en la formulación informal, y utilizan la semántica del formalismo para establecer normas precisas de interpretación que permitan superar varios problemas del lenguaje natural. Las ventajas encontradas son:

- Derivar permisos o consecuencias lógicas de la especificación para confirmar usuarios a través del teorema deductivo de las técnicas de prueba [18].
- Confirmar que una especificación operacional satisface especificaciones más abstractas, o generar contra-ejemplos de comportamiento, a través del modelo algorítmico de las técnicas de comprobación [21-23].
- Generar contra-ejemplos de las afirmaciones de una especificación declarativa [24].
- Generar escenarios concretos que ilustren características deseadas o no-deseadas de la especificación [25], o inferir inductivamente la especificación de escenarios [26].
- Producir animaciones de la especificación para comprobar si es adecuada [27, 28].
- Comprobar eficientemente las formas específicas de consistencia/completitud de la especificación [29].
- Generar excepciones de alto nivel y pre-condiciones de conflicto que puedan hacer que la especificación no se cumpla [30, 31].
- Generar especificaciones de alto nivel, como invariantes o condiciones de fortaleza [32, 34].
- Transferir refinamientos de la especificación para generar obligaciones de prueba [35-36].
- Generar casos de prueba y oráculos desde la especificación [38-39].
- Soportar la reutilización formal de los componentes a través de la especificación correspondiente [40, 41].
- Ser la base para la ingeniería inversa y para la evolución del software desde el código [42].

2.1.2 Principios de la especificación formal

Como producto de los procesos investigativos y de la experiencia del trabajo de los investigadores, en la revisión se encontraron los siguientes principios de la especificación formal:

- *No es formal*. Primero se debe determinar qué son las propiedades de estado para poderlas precisar y formalizar, por lo que es necesario formularlas en un lenguaje que las partes interesadas puedan utilizar y comprender: el lenguaje natural.

- *No tiene sentido si no cuenta con una definición informal precisa acerca de cómo interpretarla en el dominio*. Una formalización implica términos y predicados que pueden tener varios y diferentes significados, por lo que tendrán sentido si se definen con precisión mediante la asignación de nombres a las funciones/predicados y a las funciones/relaciones entre los objetos del dominio. Esta asignación debe ser precisa e informal para evitar la regresión infinita, un principio que a menudo se pasa por alto [43].

- *Es más que un simple proceso de traducción de lo informal a lo formal*. La especificación de un sistema complejo requiere objetos relevantes y fenómenos que se identifican, relacionan y caracterizan a través de propiedades compartidas. La construcción del modelo y la descripción de las propiedades están íntimamente unidos a los componentes de cualquier proceso de especificación.

- *Es difícil de desarrollar y evaluar*. Esto se debe a la diversidad y sofisticación de los errores y a la multiplicidad de opciones de modelado que se pueda tener. Como consecuencia, la especificación formal raramente es *correcta* al primer instante; sin embargo, también se señala que incluso la especificación *incorrecta* puede ayudar a encontrar problemas en la formulación original.

- *La razón para especificar las opciones de modelado es que es importante para su explicación y evolución*. Una justificación que raramente se documenta [44].

- *Los subproductos de la especificación formal frecuentemente son más importantes que ella misma.* Porque incluyen una especificación más informal que obtienen de la retroalimentación, estructuración y análisis de la expresión formal y de otros productos de menor nivel.

- *Para que un sistema formal sea útil debe tener un dominio de aplicabilidad limitado*. Para expresarlos naturalmente y para analizarlos eficientemente, los tipos específicos de sistemas requieren tipos específicos de técnicas.

2.1.3 La especificación formal en contexto

Cada año se conoce casos de éxito al utilizar especificaciones formales en sistemas reales, especialmente en trabajos de re-ingeniería [11] y de desarrollo de sistemas [78, 79]. En este último caso se reporta evidencias de que su aplicación no incrementa los costos de desarrollo, mientras que se mejora la calidad de los productos. Aunque la mayoría de estudios se aplica en los sistemas de transporte, también se reporta casos de éxito en los Sistemas de Información, los sistemas de telecomunicaciones, el control de plantas de energía, los protocolos y la seguridad.

Algunos ejemplos se pueden encontrar en [11, 13, 14], y se destaca el trabajo de Behm [79], quien describe lo sucedido en el sistema del metro de París, cuando se abrió una nueva línea con tráfico totalmente controlado por software y con trenes sin conductor. Los componentes de seguridad crítica a bordo, a lo largo de la pista y en el piso, se desarrollaron formalmente utilizando el método de máquina abstracta de B [56].

Este desarrollo incluyó modelos abstractos de los componentes, el refinamiento para modelos concretos y una traducción automatizada a código ADA. De acuerdo con el autor, el sistema posee cerca de 100.000 líneas de especificación en B, que cubren los modelos abstracto y concreto, y 87.000 líneas de código ADA. El refinamiento se validó mediante pruebas formales, la herramienta B generó automáticamente 28.000 lemas y el 65% de las normas fueron añadidas a las pruebas

de descarga. De esta forma fue posible encontrar la mayoría de errores, que se fijaron posteriormente en el desarrollo concurrente. Además, luego de desplegar el proceso convencional de pruebas no se encontraron errores diferentes.

El éxito en este estudio de caso se podría explicar porque: 1) el lenguaje B tiene una base matemática simple, que les permite a los ingenieros utilizarlo después de un período razonablemente corto de formación, 2) la técnica de especificación multi-nivel permite pasar relativamente fácil, y de manera apropiada y demostrable, desde el modelo abstracto al código, 3) el soporte metodológico se presentó en forma de directrices y heurísticas, con el objetivo de guiar los procesos de desarrollo y de validación, 4) se diseñó un modelo explícitamente para el proceso de desarrollo/validación, que se integró al de la empresa para adaptarlo a prácticas convencionales como las pruebas (la falta de integración se reconoce como un obstáculo importante para adoptar los métodos formales [74]), y 5) el proceso se soportó en herramientas robustas.

En el siglo pasado la madurez tecnológica de las herramientas de especificación se incrementó notablemente, y su eficacia logró el análisis de especificaciones formales y la derivación de información útil. El desempeño en especificaciones grandes también se incrementó y cada vez fueron más fáciles de utilizar (los animadores de especificación y los modelos de prueba fueron particularmente útiles a este respecto). Por otro lado, se originó una tendencia prometedora hacia la integración múltiple, con el objetivo de ofrecer un amplio espectro de análisis a diversos costos (el conjunto de herramientas SCR es un ejemplo de esta tendencia [73]).

2.2 Técnicas de especificación formal

Las técnicas de especificación formal reportadas en la literatura se detallan en la Tabla 1.

Tabla 1. Técnicas descritas en los trabajos primarios

Técnica	Características
Basada en historias	Describen la mayor cantidad de historias (comportamientos) en el tiempo. Las propiedades se especifican de acuerdo con las aserciones de la lógica temporal en los objetos del sistema y mediante operadores relacionados con los estados pasados, presentes y futuros, que pueden ser lineales [8] o ramificados [45]. Estas estructuras de tiempo pueden ser discretas [46, 47], densas [48], o continuas [49], y las propiedades se pueden referir a puntos de tiempo [46, 47], intervalos de tiempo [50], o a ambos [51]; frecuentemente se especifican en las fronteras del tiempo, sin embargo, las lógicas temporales son necesarias en tiempo real [52, 53].
Basadas en estados	Describen el sistema con base en los estados arbitrarios ocurridos en un lapso de tiempo. Las propiedades se especifican con invariantes que restringen los objetos del sistema, como una fotografía instantánea, o por pre y pos-aserciones que restringen su aplicación operacional en cualquier instante. Esto puede ser explícito o implícito, dependiendo de si la aserción contiene o no ecuaciones que definan constructivamente la salida. Lenguajes como Z [10, 54, 55], VDM [56] y B [36] aplican estas técnicas. También se ha propuesto las variantes orientadas por objetos [57].
Basadas en transiciones	Describen las transiciones requeridas desde un estado a otro. Las propiedades se especifican mediante un conjunto de funciones de transición en la máquina de estados, que proporciona el estado de salida correspondiente para cada estado de entrada y para cada activación del evento. La ocurrencia de un evento inicial es una condición suficiente para que tenga lugar la transición correspondiente (a diferencia de una pre-condición, captura una obligación), y las pos-condiciones se especifican para proteger la transición. Lenguajes como Statecharts [58], PROMELA [86], STeP-SPL [20], RSML [31] y SCR [73] se basan en estas técnicas.
Funcionales	Especifican el sistema como un conjunto estructurado de funciones matemáticas: 1) *Algebraica*, donde las funciones se agrupan en los tipos de objetos que aparecen en el dominio o co-dominio para definir las estructuras algebraicas, y las propiedades se especifican como ecuaciones condicionales que capturan el efecto de las funciones; lenguajes como OBJ [60], ASL [61], PLUSS [62] y LARCH [63] se basan en ellas, 2) *Funciones de orden superior*, donde las funciones se agrupan en teorías lógicas que contienen definiciones tipo (predicados lógicos), declaraciones de variables y axiomas, que definen las diversas funciones; lenguajes como HOL [64] y PVS [19, 65] se basan en este enfoque.
Operacionales	Caracterizan el sistema como una colección estructurada de procesos que puede ejecutar una máquina más o menos abstracta. Lenguajes como Paisley [66], GIST [67] y las redes Petri [68] se basan en estas técnicas.

2.2.1 Evaluación y comparación

Con base en las mismas recomendaciones de los estudios de la muestra final y en la experiencia del trabajo de los autores al respecto, para realizar la evaluación y comparación de las técnicas halladas en la revisión se definieron varios criterios. Como era de esperar, algunos son interdependientes e incluso conflictivos, por lo que una elección razonable depende de las prioridades del sistema y de la tarea en cuestión.

- *Detalle de la especificación y capacidad del código.* Cada una de las técnicas analizadas incorpora alguna tendencia semántica, como las funcionales, que se basan en estados, se centran en los comportamientos secuenciales y proporcionan estructuras adecuadas para definir objetos complejos, por lo que tienen mayor aceptación para sistemas transaccionales; o las basadas en historias, en transiciones y las operacionales, que se enfocan en comportamientos concurrentes y proporcionan solo estructuras simples para definir los objetos a manipular, por lo que su aceptación es mayor para sistemas reactivos.

  Pero existen enfoques híbridos que intentan ampliar su aplicación, como los propuestos en [69, 70]. Más allá de esta tendencia semántica, los lenguajes formales deben permitir que las propiedades del sistema se expresen sin necesidad de utilizar demasiado código, ni que la especificación se refiera a la definición de problemas y no a las soluciones de programación. Idealmente, debería existir una correspondencia simple y directa entre la formulación de una propiedad en lenguaje natural y su contraparte formal, pero raramente se da el caso.

  A diferencia del lenguaje natural, los formales tienen limitaciones, por ejemplo, en un lenguaje de primer orden es complicado hacer referencia a operaciones como argumentos de predicados, de modo que se necesita algunos trucos de codificación para superar el problema, como la introducción de eventos auxiliares. Algunos lenguajes formales son débiles para soportar las referencias temporales, las de tiempo, las explícitas o las implícitas, que frecuentemente ocurren en las formulaciones naturales; por ejemplo, cuando se incorpora inhabilidades en las especificaciones basadas en estados para referirse al pasado, por lo que cuando tal o cual evento ocurre es necesario introducir variables auxiliares en la codificación, con las correspondientes operaciones de actualización que se deben especificar en cada modificación del estado, como en la programación imperativa.

  Las técnicas basadas en historias son la principal excepción a este problema, sin embargo, también presentan inconvenientes para especificar ordenaciones relativas a eventos. En este sentido, Dwyer et al. [71] describe un ejemplo, relativamente simple, de una propiedad de pedido que requiere: *seis niveles de operador de anidación en lógica temporal lineal*. Las especificaciones algebraicas figuran entre las que requieren más experiencia en codificación, por lo que, debido a los problemas de la expresividad del lenguaje, la codificación de la especificación puede requerir mayor conocimiento especializado.

- *Capacidad de construcción y nivel de operación.* Las técnicas de especificación deberían proporcionar facilidades para construir, incrementalmente, especificaciones complejas en pequeñas partes, y los cambios locales en las características del problema se deberían reflejar en cambios locales en la especificación de requisitos. Estos requisitos dependen de: 1) mecanismos del lenguaje, para estructurar la especificación y el razonamiento composicional, y 2) la disponibilidad de un método incremental, para construir, analizar y modificar. Algunos lenguajes formales soportan mecanismos de estructuración básica para especificaciones modulares, como encapsulamiento, herencia, inclusión, generalización, enriquecimiento, entre

otras. Otros lenguajes también soportan relaciones de refinamiento como base para desarrollar y analizar la especificación incremental, por ejemplo, reificación de datos [36, 56], composición/descomposición de componentes a través de conectores lógicos [54], composición/descomposición de estados [58], o abstracción/refinamiento de objetivos [59].

- *Facilidad de uso.* Para los ingenieros de software con experiencia y un adecuado nivel de formación debería ser posible escribir especificaciones de buena calidad. En la revisión se encontró que este criterio depende de todos los anteriores más algunos otros. El lenguaje debería tener una base teórica sencilla, lo que probablemente explica la popularidad de los basados en nociones matemáticas simples y bien comprendidas, como conjuntos y relaciones y funciones [10, 19, 36, 54]. Además, también debería eximir complejidades, como la necesidad de especificaciones basadas en estados a través de axiomas de marco adicional [72].

- *Nivel de comunicación*. Contrario al lenguaje, la técnica debería ser accesible para personas razonablemente bien entrenadas, de tal manera que puedan leer y comprobar especificaciones de alta calidad. Este criterio también depende de los anteriores, particularmente de la cercanía entre la especificación y su correspondiente formulación en lenguaje natural, y del formato exterior que la especificación pueda adoptar. Esto explica la popularidad que tienen actualmente las técnicas que soportan formatos tabulares [11, 59, 65] y notaciones diagramáticas [58].

- *Calidad del análisis.* La efectividad de una técnica depende del grado en que satisfaga los objetivos planteados, porque no tiene sentido escribir especificaciones formales sino van a recibir retroalimentación de las herramientas automatizadas. Este proceso debería soportar un amplio rango de análisis en el espacio de las posibilidades, pero, con algunas excepciones notables como la descrita en [73], hasta ahora esencialmente es una ilusión. Por lo general, favorecer a uno u otro tipo de análisis permite la selección de una u otra técnica de especificación. Usualmente, cuanto más eficiente sea el análisis, más esfuerzo de codificación requiere del lado de la especificación.

  Este es el caso de la especificación basada en animaciones para ejecutar especificaciones operacionales, o para la reescritura de los términos de las algebraicas. Ilustrar el modelo a los probadores está bien, porque el lector no convencido puede ver lo que refleja su código de entrada para una aplicación compleja. Por otro lado, el análisis más completo es la intervención más experta usualmente requerida, y los asistentes de las pruebas son un buen ejemplo de esto [74]. Debería ser claro poder traducir cualquier multi-criterio de análisis en favor de un marco multi-paradigmático, en el que se integraran de forma coherente los formalismos complementarios, los métodos y las herramientas, a fin de combinar lo mejor de cada paradigma para dominios, tareas y problemas específicos. Algunos intentos preliminares se encaminan en esta dirección [75-77].

3. ANÁLISIS DE RESULTADOS

A pesar de que los resultados demuestran un progreso constante en el desarrollo de estas técnicas, también es cierto que tienen deficiencias, algunas de las cuales explican el por qué algunos las consideran no-adecuadas para la fase crítica de la especificación y el análisis de requisitos en los sistemas complejos de hoy.

- *Alcance*. La mayoría se limita a la especificación de requisitos funcionales (lo que se espera que haga el sistema) y, generalmente, dejan por fuera de cualquier tratamiento formal a los no-

funcionales. Una excepción son las técnicas que permiten propiedades de distribución para formalizar y calcular requisitos [37].

- *Separación de cometidos*. La mayoría no proporciona soporte para hacer una separación clara entre los requisitos previstos del sistema, los supuestos acerca del entorno y las propiedades del dominio de la aplicación, por lo que no se puede hacer una diferenciación entre los descriptivos y los propositivos (también llamados *indicativos* y *optativos* [43]), porque se mezclan todos en la especificación.

- *Ontologías de bajo nivel*. En el desarrollo de las soluciones los conceptos, en términos de qué problemas tienen que ser estructurados y formalizados, se tratan a nivel de programación (con mayor frecuencia datos y operaciones), pero las exigencias actuales obligan a elevar el nivel de abstracción y la riqueza conceptual encontrada en los documentos de requisitos informales, como los objetivos y sus refinamientos, los agentes y sus responsabilidades, las alternativas, y así sucesivamente [80, 81].

- *Integración*. Con algunas excepciones, estas técnicas están aisladas de otros productos y procesos del software. Esto se debe a que presentan: 1) *aislamiento vertical*, al no prestar atención a los productos que continúan en la siguiente fase del ciclo de vida del software y que dependen de la especificación formal, como objetivos, requisitos y presunciones, ni a los productos que quedaron atrás y que condujeron a la especificación formal, como los componentes arquitecturales; y 2) *aislamiento horizontal*, al no prestar atención a los productos que las acompañan y a los que debería estar vinculada la especificación formal, como la especificación informal correspondiente, la documentación de opciones, la validación de datos, la información de gestión de proyectos, entre otras.

- *Orientación*. La literatura de la especificación formal se dedica, principalmente, a describir un conjunto de notaciones y un análisis *a posteriori* de especificaciones escritas usando esas notaciones y, generalmente, no trata los métodos constructivos para diseñar de forma segura las especificaciones *correctas* para sistemas complejos, sea sistemática o incrementalmente. En lugar de proponer cada vez más lenguajes, la comunidad debería dedicar mayor esfuerzo a elaborar y validar métodos para diseñar y modificar buenas especificaciones.

- *Nivel intelectual*. Generalmente las técnicas de especificación formal requieren que el ingeniero de software posea buena experiencia en sistemas formales (lógica matemática en particular), en técnicas de análisis y en el uso de herramientas de caja blanca, pero debido a la escasez de tales profesionales y al alto costo de tenerlos, actualmente su utilización todavía es limitada en los proyectos industriales, a pesar de los beneficios que prometen.

- *Retroalimentación*. La mayoría de técnicas y herramientas de análisis formal son efectivas para señalar los problemas de los sistemas, pero, en general, son deficientes para sugerir las causas de dichos problemas y para proponer acciones de recuperación.

4. EL FUTURO DE LA ESPECIFICACIÓN FORMAL

Las técnicas y herramientas para la especificación formal que se desarrollan actualmente y que se propondrán en el futuro, deberán incorporar los siguientes requisitos y retos para que la especificación formal logre sus objetivos y para que los productos de la Ingeniería del Software logren el nivel de calidad que se espera de ellos.

1. *Enfoque constructivo.* El enfoque actual, dedicado casi exclusivamente al análisis *a posteriori* de especificaciones posiblemente pobres, se debe reemplazar por uno más constructivo, en el que la especificación se diseñe de forma incremental desde un nivel más alto. De esta forma se podrá garantizar un adecuado nivel de calidad desde la construcción del producto. Se podría hablar entonces de un método diseñado a partir de una colección de estrategias de construcción de modelos, de reglas de selección de estilos, de reglas para derivar especificaciones y de directrices y heurísticas; unas podrían ser de dominios independientes y otras de dominios específicos. El método deberá proporcionar una activa orientación en los procesos de toma de decisiones, y estar soportado por herramientas de especificación automatizadas, que presten asistencia en los puntos de decisión y que registren los procesos subsiguientes para su documentación (y posible reproducción) en caso de evolución posterior.

2. *Soporte para análisis comparativo.* Hasta el momento la investigación en especificación formal revela que diferentes especificadores, aunque posean el mismo conocimiento del sistema, pueden construir especificaciones diferentes para la formulación inicial de una solución. Esto también es cierto para el programa terminado, pero en este caso existe por lo menos un momento de verdad único: *funciona o no funciona satisfactoriamente*. Más allá de esas cualidades para la especificación se necesita criterios precisos y medidas de evaluación para comparar sus méritos relativos.

3. *Integración.* La tecnología del futuro se deberá preocupar por integrar las especificaciones, vertical y horizontalmente, al ciclo de vida del software; partiendo desde los objetivos de alto nivel para el diseño funcional de componentes arquitectónicos y desde la formulación informal, hasta la especificación formal de los productos relacionados.

4. *Nivel de abstracción.* Las técnicas deben pasar del diseño funcional a la Ingeniería de Requisitos, donde el impacto de los errores es aún más importante. Se requiere lenguajes, métodos y herramientas que soporten ontologías más robustas y que se orienten a los problemas superiores, y programas que permitan soportarlas correctamente. Los intentos preliminares incluyen al refinamiento orientado por objetivos [37, 82], el análisis de conflicto a nivel de objetivos [30, 37] y el manejo de excepciones a nivel de objetivos [31].

5. *Mecanismos estructurados.* La mayoría de las propuestas disponibles hasta el momento, para modularizar especificaciones grandes, se han copiado desde sus homólogos de la programación. Las propuestas orientadas a problemas deberían estar disponibles como, por ejemplo, puntos de vista de las partes interesadas o de las visiones del problema [83].

6. *Alcance.* Las técnicas de especificación necesitan extenderse para hacerles frente a las diferentes categorías de requisitos no-funcionales, como rendimiento, integridad, confidencialidad, exactitud de la información, disponibilidad, tolerancia a fallos, costos operativos y mantenibilidad, entre otros, que se elicitan en la Ingeniería de Requisitos, y que desempeñan un destacado papel en el diseño arquitectónico. Las técnicas de razonamiento cualitativo son un primer paso en esta dirección [82]. Categorías específicas pueden requerir lenguajes y técnicas de análisis específicos.

7. *Nivel de complejidad.* El uso de especificaciones formales no debe requerir conocimientos profundos en sistemas formales; la complejidad matemática debe estar oculta y las herramientas de análisis se deben poder utilizar como compiladores. El trabajo en especificación basada en patrones [71] es un paso muy prometedor en esta dirección, además, se puede aplicar para reutilizar pruebas y generar especificaciones [31, 37].

8. *Especificación múltiple.* Los sistemas complejos tienen múltiples facetas, y debido a que un paradigma aislado no es útil para todos los propósitos, debido a sus sesgos semánticos, se necesita *frameworks* en los que se pueda combinar múltiples paradigmas de forma semánticamente válida, de tal manera que se explote las mejores características de cada uno. Las diversas facetas necesitan estar vinculadas a través de reglas de consistencia. Los *frameworks* multi-paradigmáticos deben ser capaces de integrar a diversos lenguajes formales, los semi-formales y el natural, junto con las correspondientes técnicas y herramientas de análisis. Los intentos lingüísticos preliminares en esta dirección combinan redes semánticas, especificación basada en historias y estados, o basada en estados y en transiciones [77]. Mientras que la integración multilingüe es relativamente fácil de alcanzar con los lenguajes semi-formales, los formales plantean problemas semánticos difíciles.

9. *Multi-análisis.* Un *framework* multi-paradigmático debe soportar, opcionalmente, diferentes niveles de análisis: desde los simples, como el de nivel superficial (análisis de trazabilidad, controles estáticos de semántica y razonamiento cualitativo), hasta los más complejos, como el análisis a nivel profundo (verificación de algoritmos, razonamiento deductivo, o razonamiento inductivo a partir de ejemplos). Las opciones más complejas se deben utilizar solo cuando se requieran y cuando sean necesarias. En las primeras etapas un entorno multi-análisis también les debe permitir a los usuarios utilizar las instalaciones típicas de las herramientas CASE estándar, y llegar a fases más complejas de los métodos formales, a medida que logren confianza en su aplicación.

10. *Aplicación temprana.* Muchas técnicas requieren que la especificación esté completa para realizar el análisis necesario. En el futuro se necesitará que el análisis comience mucho antes, en la etapa temprana de la Ingeniería de Requisitos [62, 85] y que posteriormente se incremente. Esto garantizará la recuperación anticipada de la inversión e incrementará las utilidades (un importante objetivo señalado en [14]). Por otro lado, las técnicas deductivas también asumen que la especificación es consistente para obtener información útil, que se deriva especialmente en el contexto de la Ingeniería de Requisitos, donde se puede inferir desde puntos de vista conflictivos. Para derivar tal información se necesita sistemas formales y técnicas de razonamiento, a pesar de las inconsistencias temporales [84].

11. *Reutilización.* Es probable que los problemas en el dominio se asemejen a las soluciones, por lo que la reutilización de la especificación deberá ser más prometedora que la reutilización del código. Sorprendentemente, las técnicas para recuperar, adaptar y consolidar especificaciones reciben relativamente poca atención en la literatura. La reutilización de especificaciones, que resulten ser buenas y eficaces para sistemas similares, se deberá abordar con enfoques constructivos para la especificación del futuro.

12. *Métricas*. Para que en el futuro la especificación formal logre mejores y mayores objetivos, deberá ser posible medir los beneficios de utilizarla en la Ingeniería del Software, a través de métricas similares a las utilizadas para medir el incremento de la productividad del software. De acuerdo a los resultados en la revisión realizada, esta cuestión recibe poca atención en los procesos investigativos.

5. CONCLUSIONES

El software se ha convertido en un producto tecnológico imprescindible para la sociedad del siglo XXI, por lo que cada vez más se necesitan productos con mayor calidad, y las especificaciones formales ofrecen un amplio espectro de posibles caminos para alcanzar ese objetivo. Por este y

otros aspectos, la especificación formal debería recibir mayor atención en el mundo académico e industrial. Sin embargo, existe un extenso camino por recorrer antes que se pueda utilizar completamente en los procesos de la Ingeniería del Software. Entre los retos que plantea su utilización se requiere otros factores críticos para lograr el éxito que se espera de ella en el futuro:

- La prestación de asistencia constructiva en el desarrollo de la especificación, el análisis y la evolución.
- La integración vertical y horizontal de las especificaciones formales dentro del ciclo de vida del software.
- Abstracciones de alto nivel para la especificación y el análisis de requisitos.
- La disponibilidad de técnicas formales para requisitos no-funcionales.
- Las interfaces livianas para la especificación y el análisis multi-paradigmático.
- En lugar de solo señalar los problemas, las herramientas del futuro deberán ayudar a resolverlos.

Actualmente, los métodos formales son una realidad, y la industria y la comunidad de investigadores continúan desarrollando técnicas de especificación formal que, en el futuro, podrán funcionar adecuadamente y ofrecerán las ventajas que se espera de ellas, incluso para aplicaciones críticas.

Desde las técnicas formales ha surgido buenas prácticas de especificación y desarrollo de software, y no se pueden dejar de lado ni suplantar, el objetivo es mejorarlas.

Los futuros trabajos deberán reunir los principios de la automatización total de las pruebas, determinar las formas en que la formalidad se puede utilizar con mayor efectividad y buscar que los procesos formativos la incluyan como parte de los contenidos curriculares en Ciencias Computacionales.

Los desarrolladores y la industria deben comenzar a ver a los métodos formales, y por ende a la especificación formal, como una inversión y trabajar para garantizar que el producto logre la calidad que el usuario espera por su valor.

REFERENCIAS

[1] Randell B. (1973). The Origin of Digital Computers. Springer.
[2] Floyd R. (1967). Assigning meanings to programs. Mathematical aspects of Comp. Science 19, 19-32.
[3] Hoare C.A.R. (1969). An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576-583.
[4] Naur P. (1969). Proofs of algorithms by General Snapshots. BIT 6, 310-316.
[5] Dijkstra E.W. (1975). Guarded commands, non-determinacy and the formal derivation of programs. Communications of the ACM 18(8), 453-457.
[6] Parnas D. (1972). A Technique for Software Module Specification with Examples. Communications of the ACM 15(5), 330-336.
[7] Liskov B. y Zilles S. (1975). Specification Techniques for Data Abstractions. IEEE Transactions on Software Engineering 1(1), 7-18.
[8] Pnueli A. (1977). The Temporal Logics of Programs. En 18th IEEE Symposium on Foundations of Computer Science. Rhode Island, USA.
[9] Parnas D. (1977). The Use of Precise Specifications in the Development of Software. En IFIP Congress. Toronto, Canada.

[10] Abrial J. (1980). The Specification Language Z: Syntax and Semantics. Oxford University Press.
[11] Heninger K. (1980). Specifying Software Requirements for Complex Systems: New Techniques and their Application. IEEE Transactions on Software Engineering 6(1), 2-13.
[12] Wing J. (1990). A Specifier's Introduction to Formal Methods. IEEE Computer 23(9), 8-24.
[13] Hinchey M. y Bowen, J. (1995). Applications of Formal Methods. Prentice-Hall.
[14] Clarke E. y Wing J. (1996). Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28(4), 626-643.
[15] Serna E. (2010). Formal Methods and Software Engineering. Rev. Virtual Universidad Católica del Norte 30, 158-184.
[16] Swartout W. y Balzer R. (1982). On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM 25(7), 438-440.
[17] Yue K. (1987). What Does It Mean to Say that a Specification is Complete? En Fourth Inter-national Workshop on Software Specification and Design. Monterey, USA.
[18] Meyer B. (1985). On Formalism in Specifications. IEEE Software 2(1), 6-26.
[19] Owre S. et al. (1995). Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. IEEE Transactions on Software Engineering 21(2), 107-125.
[20] Manna Z. y the STep Group. (1996). STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems. Lecture Notes in Computer Science 1102, 415-418.
[21] Clarke E. y Emerson E. (1986). Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Progra. Languages and Systems 8(2), 244-263.
[22] Atlee J. (1993). State-Based Model Checking of Event-Driven System Requirements. IEEE Transactions on Software Engineering 19(1), 24-40.
[23] Heitmeyer C. et al. (1998). Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications. IEEE Transactions on Software Engineering 24(11), 927-948.
[24] Jackson D. y Damon C. (1996). Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. IEEE Transactions on Software Engineering 22(7), 484-495.
[25] Hall R. (2000). Explanation-Based Scenario Generation for Reactive System Models. Automated Software Engineering 7(2), 157-177.
[26] Lamsweerde A. y Willemet L. (1998). Inferring declarative requirements specifications from operational scenarios. IEEE Transactions on Software Engineering 56, 1089-1114.
[27] Harel D. et al. (1990). STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering 16(4), 403-414.
[28] Thompson J. et al. (1999). Specification-Based Prototyping for Embedded Systems. En ESEC/FSE'99. Toulouse, France.
[29] Heimdahl M. y Leveson N. (1996). Completeness and Consistency in Hierarchical State-Based Requirements. IEEE Transactions on Software Engineering 22(6), 363-377.
[30] Lamsweerde A. (1998). Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Transactions on Software Engineering 24(11), 908-926.
[31] Lamsweerde A. y Letier E. (2000). Handling Obstacles in Goal-Oriented Requirements Engineering. IEEE Transactions on Software Engineering 26(10), 978-1005.
[32] Lamsweerde A. y Sintzoff M. (1979). Formal Derivation of Strongly Correct Concurrent Programs. Acta Informatica 12(1), 1-31.
[33] Park D. et al. (1998). Static Analysis to Identify Invariants in RSML Specifications. En 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. Lyngby, Denmark.
[34] Jeffords R. y Heitmeyer C. (1998). Automatic Generation of State Invariants from Requirements Specifications. ACM SIGSOFT Software Engineering Notes 23(6), 56-59.
[35] Morgan C. (1990). Programming from Specifications. Prentice-Hall.
[36] Abrial J. (1996). The B-Book: Assigning Programs to Meanings. Cambridge University Press.
[37] Darimont R. y Lamsweerde A. (1996). Formal Refinement Patterns for Goal-Driven Requirements Elaboration. ACM SIGSOFT Software Engineering Notes 21(6), 179-190.
[38] Bernot G. et al (1991). Software Testing Based on Formal Specifications: A Theory and a Tool. Software Engineering Journal 6(6), 387-405.
[39] Richardson D. et al. (1992). Specification-based test oracles for reactive systems. International Conference on Software Engineering. Melbourne, Australia.

[40] Reubenstein H. y Waters R. (1991). The Requirements Apprentice: Automated Assistance for Requirements Acquisition. IEEE Transactions on Software Engineering 17(3), 226-240.
[41] Massonet P. y Lamsweerde A. (1997). Analogical Reuse of Requirements Frameworks. En Third IEEE International Symposium on Requirements Engineering. Annapolis, USA.
[42] Gannod G. y Cheng, B. (1996). Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering. Journal of Automated Software Engineering 3, 139-164.
[43] Zave P. y Jackson M. (1997). Four Dark Corners of Requirements Engineering. ACM Transactions on Software Engineering and Methodology 6(1), 1-30.
[44] Souquières J. y Levy N. (1993). Description of Specification Developments. En First IEEE Symposium on Requirements Engineering. San Diego, USA.
[45] Emerson E. y Halpern J. (1986). Sometimes and not Never Revisited: On Branching versus Linear Time Temporal Logic. Journal of the ACM 33(1), 151-178.
[46] Manna Z. Pnueli A. (1992). The Temporal Logic of Reactive and Concurrent Systems. Springer.
[47] Lamport L. (1994). The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems 16(3), 872-923.
[48] Greenspan S. et al. (1986). A Requirements Modeling Language and its Logic. Information Systems 11(1), 9-23.
[49] Hansen K. et al. (1993). Specifying and Verifying Requirements of Real-Time Systems. IEEE Transactions on Software Engineering 19(1), 41-55.
[50] Moser L. et al. (1997). A Graphical Environment for the Design of Concurrent Real-Time Systems. ACM Transactions on Software Engineering and Methodology 6(1), 31-79.
[51] Jahanian F. y Mok A. (1986). Safety Analysis of Timing Properties in Real-Time Systems. IEEE Transactions on Software Engineering 12, 890-904.
[52] Koymans R. (1992). Specifying message passing and time-critical systems with temporal logic. Springer.
[53] Dubois E. et al. (1991). A Formal Language for the Requirements Engineering of Computer Systems. En A. Thayse (Ed.), Introducing a Logic Based Approach to Artificial Intelligence (pp. 357-433). John Wiley.
[54] Spivey J. (1992). The Z Notation - A Reference Manual. Prentice-Hall.
[55] Potter B. et al. (1996). An Introduction to Formal Specification and Z. Prentice-Hall.
[56] Jones C. (1990). Systematic Software Development using VDM. Prentice-Hall.
[57] Lano K. (1995). Formal Object-Oriented Development. Springer.
[58] Harel D. (1987). Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 231-274.
[59] Parnas D. y Madey J. (1995). Functional Documents for Computer Systems. Science of Computer Programming 25(1), 41-61.
[60] Futatsugi K. et al. (1985). Principles of OBJ. En ACM Symposium on Principles of Programming Languages. New Orleans, USA.
[61] Astesiano E. (1986). An introduction to ASL. Universitat Passau.
[62] Gaudel M. (1992). Structuring and Modularizing Algebraic Specifications: The PLUSS specification language, evolutions and perspectives. Lecture Notes in Computer Science 557, 1-18.
[63] Guttag J. y Horning J. (1993). LARCH: Languages and Tools for Formal Specification. Springer.
[64] Gordon M. y Melham T. (1993). Introduction to HOL. Cambridge University Press.
[65] Crow J. et al. (1995). A Tutorial Introduction to PVS. En Workshop on Industrial-Strength Formal Specification Techniques. Boca Raton, USA.
[66] Zave P. (1982). An Operational Approach to Requirements Specification for Embedded Systems. IEEE Transactions on Software Engineering 8(3), 250-269.
[67] Balzer R. et al. (1982). Operational Specification as the Basis for Rapid Prototyping. ACM SIGSOFT Software Engineering Notes 7(5), 3-16.
[68] Milner R. (1989). Communication and Concurrency. Prentice-Hall.
[69] Faulk S. et al. (1992). The CORE Method for Real-Time Requirements. IEEE Software 9(5), 22-33.
[70] George A. et al. (1995). The RAISE Development Method. Prentice-Hall.
[71] Dwyer M. et al. (1999). Patterns in Property Specifications for Finite-State Verification. En 21th Intrnational Conference on Software Enginering. Los Angeles, USA.

[72] Borgida A. et al. (1995). On the Frame Problem in Procedure Specifications. IEEE Transactions on Software Engineering 21(10), 785-798.
[73] Heitmeyer C. et al. (1998). SCR*: A Toolset for specifying and Analyzing Software Requirements. En 10th Annual Conference on Comuter-Aided Verification. Vancouver, Canada.
[74] Craigen D. et al (1995). Formal Methods Technology Transfer: Impediments and Innovation. En M. Hinchey y J. Bowen (Eds.), Applications of Formal Methods (pp. 399-419). Prentice-Hall.
[75] Niskier C. et al. (1989). A Pluralistic Knowledge-Based Approach to Software Specification. En 2nd European Software Engineering Conference. Coventry, UK.
[76] Zave P. y Jackson M. (1993). Conjunction as Composition. ACM Transactions on Software Engineering and Methodology 2(4), 379-411.
[77] Zave P. y Jackson M. (1996). Where Do Operations Come From? A Multiparadigm Specification Technique. IEEE Transactions on Software Engineering 22(7), 508-528.
[78] Hall A. (1996). Using Formal Methods to Develop an ATC Information System. IEEE Software 12(6), 66-76.
[79] Behm P. et al. (1989). Météor: A Successful Application of B in a Large Project. En Wold Congress on Formal Methods in the Development of Computing Systems. Toulouse, France.
[80] Feather M. (1987). Language Support for the Specification and Development of Composite Systems. ACM Transactions on Programming Languages and Systems 9(2), 198-234.
[81] Mylopoulos J. (1998). Information Modeling in the Time of the Revolution. Information Systems 23(3-4), 127-155.
[82] Mylopoulos J. et al. (1992). Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. IEEE Transactions on Software Engineering 18(6), 483-497.
[83] Jackson D. (1995). Structuring Z Specifications with Views. ACM Transactions on Software Engineering and Methodology 4(4), 365-389.
[84] Hunter A. y Nuseibeh B. (1998). Managing Inconsistent Specifications: Reasoning, Analysis and Action. ACM Transactions on Software Engineering and Methodology 7(4), 335-367.
[85] Serna E. (2021) Gestión de la Ingeniería de Requisitos integrando principios del Pensamiento Complejo. Editorial Instituto Antioqueño de Investigación.
[86] Vlaovic B. et al. (2007). Automated Generation of Promela Model from SDL Specification. Computer Standards & Interfaces 29(4), 449-461.

CAPÍTULO XI
La Investigación en Verificación Formal - Un estado del arte[1]

Edgar Serna M.[1]
David Morales V.[2]
[1]Instituto Antioqueño de Investigación
[2]Diversien S.A.S.

Para hacerle frente a la creciente complejidad de los sistemas en el siglo XXI, la investigación en verificación formal de hardware y software ha logrado progresos en el desarrollo de metodologías y herramientas. La función explícita de esta Verificación es encontrar errores y mejorar la confianza en el diseño de los sistemas, lo que supone un reto para la Ingeniería del Software actual. El objetivo de esta investigación es realizar una revisión sistemática a la literatura para determinar el estado del arte de la investigación en verificación formal e identificar los enfoques, métodos, técnicas y metodologías empleadas, lo mismo que la intensidad de la misma. En el proceso se encontró que la investigación en esta área ha progresado desde 2005, que hasta el momento mantiene un número promedio de investigaciones año tras año y que predomina la aplicación en sistemas de control e interacción. Además, que el estudio de casos es el método más utilizado y que la investigación empírica es la más aplicada.

[1] Presentado en Informática 2013, XV Convención y Feria Internacional. La Habana, Cuba. 2013.

INTRODUCCIÓN

La verificación funcional se ha convertido en el cuello de botella para el diseño de sistemas complejos, porque simular los diseños es costoso en términos de dinero y de tiempo, y una simulación completa es prácticamente imposible. Actualmente y como respuesta a estos problemas, los diseñadores utilizan los métodos formales para realizar la verificación formal en algunos de sus productos. Pero aún persiste una amplia brecha para la verificación de los grandes diseños, en parte por los inconvenientes de verificar completamente debido a la complejidad de los problemas que tratan [1]. Esto ha ocasionado que en muchos países la academia, la industria y el gobierno se enfrenten al reto de reducir esa brecha tecnológica, y a que se propongan nuevas e ingeniosas soluciones para la especificación, el diseño, la estructuración y la aplicación de casos de prueba del software mediante verificación formal.

Por otra parte, la verificación funcional es un elemento crítico en el desarrollo de los actuales y complejos Sistemas de Información. La ley de Moore todavía se aplica al crecimiento de la complejidad de los productos hardware y software, pero la complejidad de la verificación es más complicada y, de hecho, en teoría aumenta exponencialmente con la complejidad del producto y se duplica de la misma forma con el tiempo. En la comunidad de las Ciencias Computacionales se reconoce que la verificación funcional es un importante obstáculo para una metodología de diseño, y que consume hasta el 70% del tiempo de desarrollo y de recursos. Pero, incluso con esa significativa cantidad de esfuerzos y de recursos aplicada a la verificación, los defectos funcionales continúan como causa del amplio número de errores del producto final. En casos extremos los errores son artefactos de la simulación, porque no se detectan debido a la naturaleza no-exhaustiva de la verificación basada en simulación. La realidad es que no importa cuánto tiempo lleve la simulación, ni que tan exhaustivo sea el plan de pruebas, todo intento de validar un diseño mediante simulación es de por sí incompleto para cualquier sistema.

La verificación formal es un proceso sistemático, que utiliza razonamiento matemático para verificar que la especificación del diseño se conserva en la implementación. Con esta Verificación es posible superar los desafíos de la simulación, porque se puede explorar, de forma algorítmica e exhaustiva, todos los posibles valores de entrada. En otras palabras, para lograr un alto grado de observación del producto no es necesario exagerar el diseño o crear escenarios múltiples.

Uno de los objetivos de la verificación formal es garantizar la completa cobertura del espacio de los estados en el diseño que se prueba, para lo que utiliza y aplica técnicas como la verificación de modelos mediante la exploración del espacio de estados, y técnicas automatizadas para probar los teoremas. Actualmente, la técnica de verificación formal con mayor automatización y aceptación es Symbolic Model Verifier SMV y, aunque logra éxito como método importante para la verificación formal de diseños comerciales secuenciales, todavía es limitada con relación al tamaño de los diseños verificables [2].

La verificación formal requiere que los ingenieros piensen de forma diferente, por ejemplo, la simulación es empírica, es decir, que utilizar la prueba y el error para probar todas las posibles combinaciones y tratar de descubrir los errores puede tomar una buena cantidad de tiempo. Por lo tanto, no se logra completamente. Además, dado que los ingenieros tienen que definir y generar un alto número de escenarios de entrada, centran sus esfuerzos en cómo *romper* el diseño y no en lo que el diseño *tiene que hacer*. Por el contrario, la verificación formal es matemática y exhaustiva, y permite que el ingeniero se centre únicamente en encontrar cuál es el correcto comportamiento del diseño.

El objetivo de esta investigación es realizar una revisión sistemática de la literatura en el tema de la investigación en verificación formal, para determinar los enfoques, métodos, técnicas y metodologías de investigación empleados, y la intensidad de esa investigación. Para lograrlo se empleó el paradigma de investigación basado en la evidencias. La posibilidad de emplear este paradigma se propone en [6, 7], y tiene como objetivo identificar una pregunta que sea posible responder, que ofrezca información y que encuentre evidencias que la respondan y evalúen [8]. De acuerdo con esto, una revisión sistemática de la literatura constituye el primer paso para la realización de investigaciones basadas en evidencias. Las directrices para la realización de una revisión sistemática a la literatura se explican detalladamente en [8, 9].

1. MÉTODO

Realizar una revisión sistemática de la literatura se puede dividir en tres fases principales [8]: 1) planificación, 2) realización, y 3) documentación, que a su vez se dividen en una combinación de otros procedimientos más simples, como se representa en la Tabla 1.

Tabla 1. Fases de una revisión sistemática [9]

Fases	Procedimientos
Planificación	Especificar las preguntas de investigación Desarrollar protocolo de revisión Validar protocolo de revisión
Realización	Identificar las investigaciones relevantes Seleccionar los estudios primarios Valorar la calidad de los estudios Extraer los datos requeridos Sintetizar los datos
Documentación	Escribir el reporte de la revisión Validar el reporte

De acuerdo con [9] y [10] planear una revisión sistemática consiste en estructurar seis definiciones:

1. Las preguntas de investigación
2. El proceso de búsqueda
3. Los criterios de inclusión y exclusión
4. La valoración de la calidad
5. La recopilación de datos
6. El análisis de datos

1.1 Preguntas de investigación

Las preguntas de investigación en el desarrollo de esta investigación fueron:

P1: ¿En qué áreas de la verificación formal se investiga actualmente?
P2: ¿Cuál metodología de aplicación es la más investigada?
P3: ¿En qué técnica de verificación formal se investiga con mayor frecuencia?
P4: ¿Qué enfoques y métodos de investigación son los más utilizados?
P5: ¿Cuál es la intensidad de la investigación en verificación formal?

Con el objetivo de responder a P1, P2, P3 y P4, se asoció cada estudio primario con un enfoque o método de investigación, con una técnica y metodología aplicada y con un área cubierta. Con

respecto a P5 y para establecer las cifras que indicaran la intensidad de la actividad investigativa, se identificó un corpus de investigación del número de publicaciones en la primera década del siglo. La pendiente de la línea para la verificación formal se comparó con la pendiente correspondiente a la línea que representa la actividad de investigación en verificación funcional.

1.2 Proceso de búsqueda

Una revisión sistemática sobre un tema específico debe identificar y resaltar las fuentes específicas acerca del objeto de estudio, sin embargo, en el dominio de la verificación formal no se encontraron estas fuentes, porque los estudios relacionados se pueden publicar en revistas y conferencias, que están relacionadas tanto con la verificación funcional como con los métodos formales. El objetivo de la búsqueda fue identificar los estudios primarios que se podrían incluir o excluir del conjunto final de estudios de la revisión. El plan involucró una búsqueda automatizada en las bibliotecas ACM Digital Library, IEEE Digital Library, WOS y SpringerLink. Los parámetros de la búsqueda automatizada y su ubicación en el estudio fueron los siguientes:

- *Formal Verification*: en el título. Para todas las preguntas de investigación.
- *Discret Mathematical, Declarative Language, Formal Language, Formal Method, Formal Specification* y *Formal Verification*: en el *abstract* o el contenido. Para P1.
- *Experimentation, Case Study, Stochastic* y *Heuristic*: en el *abstract* o el contenido. Para P2.
- *Peer, Animation, Simulation, Agil Methods* y *XP*: en el *abstract* o el contenido. Para P3.
- La observación de los resultados para P1, P2 y P3 permitió clasificar el enfoque y el método de investigación para P4. Para la investigación empírica se hizo una búsqueda de los términos *Experiment, Survey, Case Study, Empirical Research* en el *abstract* y el contenido.
- *Formal Verification AND Research*: en el título. Para P5.

El total de trabajos que arrojó esta búsqueda fue 552, sin embargo, la mayoría se identificó mediante relación marginal y como resultado de la combinación de algunas de las palabras clave. La exclusión de los irrelevantes se llevó a cabo manualmente, siguiendo los criterios de inclusión y exclusión que se definen a continuación.

1.3 Criterios de inclusión y exclusión

Los trabajos seleccionados como estudios primarios debían ser relevantes para la temática de investigación, por lo que se aplicó el proceso de filtrado propuesto en [11]:

1. *Identificar los estudios relevantes*. Se consideraron solo trabajos completos publicados en *journals, full conference-congress* y *workshop*, y se descartaron s*hort paper, extended abstract* y *posters*. Se excluyeron 131 estudios.
2. *Excluir estudios con base en el título*. El criterio de exclusión aplicado fue el filtro *AND* en la búsqueda avanzada de cada librería digital. Se excluyeron 28.
3. *Excluir estudios con base en los resúmenes*. Se excluyeron 49 trabajos.
4. De los estudios resultantes *seleccionar los más relevantes para la temática de investigación* con base en el texto completo. Se decidió incluir solamente los trabajos que estuvieran estrechamente relacionados con la verificación formal. Bajo este criterio se excluyeron 145 trabajos, lo que arrojó una muestra final de 199 documentos como estudios primarios.

1.4 Valoración de la calidad

El objetivo de esta fase es validar que los estudios primarios seleccionados tengan solidez en cuanto a metodología y resultados. Teniendo en cuenta los altos estándares del proceso de revisión en las revistas y en las bases de datos seleccionadas se concluyó, con base en la evidencia, que los estudios primarios seleccionados presentan buena calidad.

1.5 Recopilación de datos

Luego de culminar el proceso de inclusión-exclusión se estructuró el conjunto de datos de los estudios primarios. Durante esta fase se recopilaron los siguientes atributos:

1. Tipo de evento: *Journal, Conference-Congress, Workshop*.
2. Publicado en: *Journal, Proceedings*.
3. Casa editor: ACM, IEEE, Springer, Elsevier.
4. Año de publicación.
5. País.
6. Clasificación del enfoque y el método. De acuerdo con Glass et al. [3], los principales enfoques investigativos científicos son: descriptivo, explicativo y empírico y, de acuerdo con Dyba y Dingsoyr [4] y Wohlin et al. [5], existen tres métodos de investigación utilizados para evaluar técnicas, métodos y herramientas: encuesta, estudio de caso y experimento.
7. Clasificación del área. Las áreas seleccionadas para la investigación fueron: modelos matemáticos, lenguajes formales, modelos automatizados, lenguajes declarativos, métodos formales y especificación formal.
8. Clasificación de la metodología. Las metodologías analizadas fueron: experimentación, estudio de casos, estocástica y heurística.
9. Clasificación de la técnica. Los estudios primarios seleccionados se clasificaron de acuerdo con el tratamiento dado en la técnica empleada: pares, animación, simulación y métodos ágiles.

Para responder a P5 se incluyeron tres tipos de trabajos de acuerdo con la siguiente clasificación:

- *De investigación científica y tecnológica*. Documento que presenta de manera detallada los resultados originales de proyectos de investigación terminado. Su estructura generalmente es: introducción, método, resultados, análisis de resultados y conclusiones.
- *De reflexión*. Documento que presenta resultados de investigaciones terminadas, o puntos de vista, desde una perspectiva analítica, interpretativa o crítica sobre un tema específico, y recurriendo a fuentes originales.
- *De revisión*. Documento el que se analiza, sistematiza e integra resultados de investigaciones publicadas o no publicadas sobre un campo en ciencia o tecnología, con el objetivo de divulgar los avances y las tendencias en desarrollo. Se caracteriza por presentar una cuidadosa revisión bibliográfica de por lo menos 50 referencias.

1.6 Análisis de datos

Los estudios primarios se tabularon y analizaron estadísticamente con el objetivo de encontrar:

1. Número de trabajos publicados por año: P5.
2. Número de trabajos publicados en *journals* y *proceedings*: P5.
3. Número de estudios por país: P5.
4. Principales temas cubiertos en verificación formal: P5.
5. Enfoque y método de Investigación: P4.
6. Área de la verificación formal en la que se investiga: P1.
7. Metodología de aplicación: P2.
8. Técnica utilizada: P3.

2. RESULTADOS Y ANÁLISIS

Con el objetivo de comprender las categorías que se asigna a cada estudio se tabularon las características del conjunto de datos de los estudios primarios. Es importante apreciar la diferencia que existe entre *actividad de investigación* y *artículo de investigación*. La primera comprende el conjunto de trabajos relevantes que fueron incluidos con base en el título, es decir, de investigación, de reflexión y de revisión, mientras que los de investigación son el resultado final de la aplicación de los criterios de inclusión y exclusión. En la Tabla 2 se presenta el dinamismo de la actividad de investigación por año y tipo de evento.

Tabla 2. Dinamismo de la investigación en verificación formal

Año	*Conference/Congress*	*Journal*	*Workshop*	Total
2000	0	3	0	3
2001	0	3	0	3
2002	1	13	0	14
2003	0	8	0	8
2004	0	3	0	3
2005	11	21	0	32
2006	8	13	8	29
2007	13	11	3	27
2008	11	14	1	26
2009	10	8	5	23
2010	11	9	7	27
2011	0	4	0	4
Total	65	110	24	199

De acuerdo con estos resultados la investigación en verificación formal se duplicó a partir del 2005, manteniendo un constante número de publicaciones hasta el momento. En la Figura 1 se presenta la comparación entre las actividades de investigación en Ingeniería del Software vs verificación formal. Los temas que abarca la investigación en la primera son diversos, entre los que se encuentra la verificación formal, pero para este análisis se tomó como un concepto aparte debido a los intereses de la investigación.

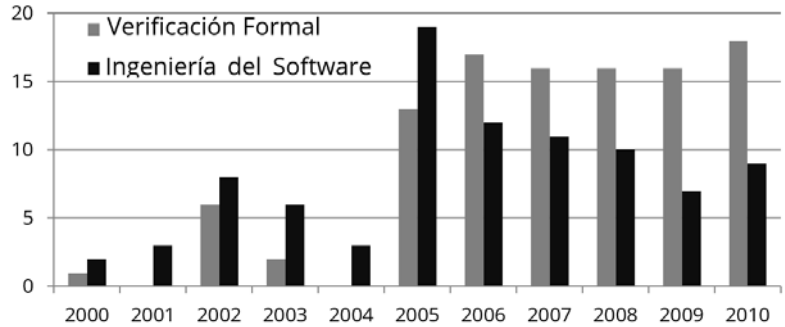

Figura 1. Intensidad de la investigación en Ingeniería del Software y la verificación formal

Al examinar las actividades relacionadas con la investigación en verificación formal por país se observa que EE.UU. aporta más de la mitad del total de publicaciones, con un 55%. Le siguen Reino Unido, Japón, China y Alemania. En los demás países, con alguna representatividad, se encontró que el interés por la verificación formal empezó un poco más tarde que en los anteriores. En la Tabla 3 se presenta la comparación entre los trabajos que reportan investigación y otro tipo de reporte relacionado con la temática de la verificación formal.

Tabla 3. Artículos de investigación vs otros artículos

Año	Investigación	Otro	% Investigación
2000	1	2	33%
2001	0	3	0%
2002	6	8	43%
2003	2	6	25%
2004	0	3	0%
2005	13	19	41%
2006	17	12	59%
2007	16	11	59%
2008	16	10	62%
2009	16	7	70%
2010	18	9	67%
2011	0	4	0%
Total	105	94	89.5%

Debido a que el objetivo de esta revisiónes averiguar métodos, técnicas y metodologías que aplican las investigaciones en verificación formal, en el resto del documento se trabaja solo con los 105 trabajos que difunden resultados de investigación. La Tabla 4 contiene las publicaciones en las que se encontraron los documentos de las investigaciones relacionadas con verificación formal y la cantidad de trabajos publicados en la línea de tiempo cubierta en esta investigación. En la Figura 2 se detalla la relación de países más activos en investigación en verificación formal.

Tabla 4. Medio y trabajos publicados en verificación formal

Medio	
Electronic Notes in Theorical Computer Science (ENTCS)	20
Formal Methods in System Design	12
IEEE Transactions on Software Engineering	8
IEEE Transactions on Systems, Man, and Cybernetics	7
International Journal on Software Tools for Technology Transfer(STTT)	6
Theorical Computer Science	4
IEEE Design y Test	6
Journal of Automated Reasoning	3
Formal Aspects of Computing	3
Computers in industry	2
Computer Standards y Interfaces	2
Science of Computer Programming	2
Journal of Systems Architecture: the Euromicro Journal	2
Real-Time Systems	2
Computer	1
IEEE Transactions on Computers	1
IEEE Software	1
IBM Journal of Research and Development	1
Journal of Computing Science in Colleges	1
Journal of Systems and Software	1
Journal of the ACM (JACM)	1
Journal of Parallel and Distributed Computing Systems	1

Future Generation Computer Systems	1
Journal of Symbolic Computation	1
Automation and Remote Control	1
Advances in Engineering Software	1
Journal of Electronic Testing: Theory and Applications	1
Environmental Modelling y Software	1
Integration, the VLSI Journal	1
International Journal of Parallel Programming	1
Programming and Computing Software	1
Nordic Journal of Computing	1
Informatics	1
Journal of Visual Languages and Computing	1
IEEE Transactions on Dependable and Secure Computing	1
EURASIP Journal on Embedded Systems	1
Annals of Software Enginnering	1
IEEE Transactions on Information Forensics and Security	1
Software Testing, Verification y Reliability	1
International Journal of Agent-Oriented Software Engineering	1

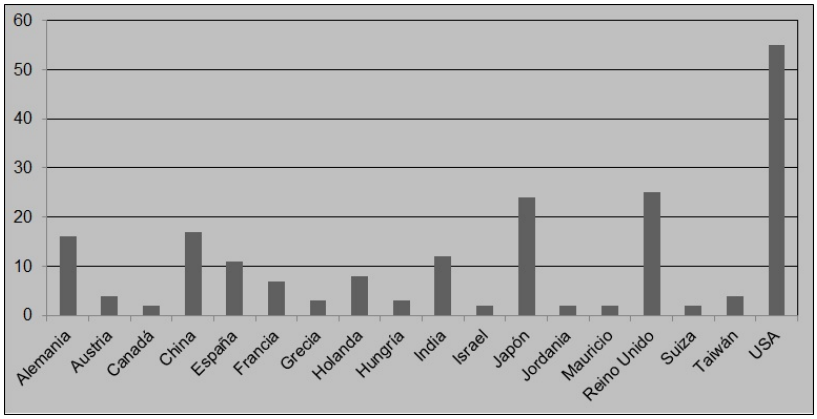

Figura 2. Actividad investigativa en verificación formal por país

La Tabla 5 presenta una comparación entre el número de universidades y el de empresas que realizan investigación en verificación formal, y el número de trabajos publicados.

Tabla 5. Universidades y empresas que investigan en verificación formal

	Cantidad	Publicaciones
Universidades	56	86
Industrias	14	19

En las Tablas 6 y 7 se detallan los resultados del análisis en cuanto a la clasificación de los enfoques y métodos de investigación en verificación formal encontrados.

Tabla 6. Métodos de Investigación

Método	Relación
Estudio de Caso	86/105
Experimento	19/105

Tabla 7. Enfoque de Investigación

Enfoque	Cantidad
Aplicado	0
Descriptivo	0
Empírico	105
Exploratorio	0

El método con mayor aplicación es el de estudio de casos, lo que refuerza el resultado de sean las universidades las que mayor participación tienen en las investigaciones de esta área. Los trabajos de investigación sobre Verificación Formal emplean un enfoque de investigación empírica, esto se debe a la necesidad de comprobación del método y al modelo aplicado. En la Tabla 8 se muestra los resultados concernientes al área de la investigación en Verificación Formal, teniendo en cuenta que son incluyentes.

Tabla 8. Áreas de Investigación en Verificación Formal

Área	Cantidad	Porcentaje
Modelo Matemático	105	100%
Modelo Automatizado	26	25%
Métodos Formales	105	100%
Especificación Formal	104	99%
Lenguajes Formales	97	92%
Lenguajes Declarativos	6	6%

Las áreas en las que más se trabaja son la especificación formal, los modelos matemáticos y los métodos formales. Esta última permite describir las propiedades del sistema a través de la matemática rigurosa, para lo cual se aplica un lenguaje de especificación formal con el que es posible especificar la funcionalidad de un programa; esto se debe a la forma como se construye la verificación formal: inicialmente se centra en la especificación, luego se construye el modelo de prueba y posteriormente se comprueba la verificación en el estudio de caso. El proceso es complejo e involucra varias herramientas, unas manuales otras automatizadas. La Tabla 9 muestra un comparativo entre las metodologías empleadas para la Verificación Formal en los trabajos analizados.

Tabla 9. Metodologías de Investigación

Metodología	Cantidad	Porcentaje
Experimental	18	17%
Estudio de Caso	81	77%
Estocástica	6	6%
Heurística	0	0%

Debido a que el método de los estudios de casos se emplea para aplicar la verificación formal y para comprobar los resultados manualmente, también aparece como la metodología predominante para validar resultados en el enfoque empírico. La parte experimental se evidencia en la participación de las investigaciones industriales. La Tabla 10 presenta las técnicas utilizadas para investigar en Verificación Formal de los estudios primarios.

Tabla 10. Técnicas de Investigación

Técnica	Cantidad	Porcentaje
Por Pares	2	2%
Animación	0	0%
Simulación	103	98%
Métodos Ágiles	0	0%

Las técnicas actuales de desarrollo se adaptan de mejor forma a los nuevos paradigmas y existe herramientas comerciales que soportan el mejoramiento de la calidad del software. A medida que los sistemas de información incrementan su complejidad, las pérdidas causadas por fallas son cada vez mayores. El 98% de los trabajos de investigación describe técnicas de simulación, esto con el fin de controlar las variables de entrada y las respuestas o salidas esperadas en los

ambientes de prueba. Llama la atención el hecho de que solo el 2% empleó la técnica de comprobación por pares, que en las revisiones a la literatura de finales de siglo era la más empleada.

3. AMENAZAS Y LIMITACIONES A LA INVESTIGACIÓN

En esta investigación se llevó a cabo una minuciosa revisión de la literatura a partir de la obtención de 199 trabajos diferentes, incluyendo algunos estudios secundarios, donde se utilizaron las referencias en el estudio primario para encontrar otros estudios. Sin embargo, se oserba que no es posible garantizar que se capturaron todos los trabajos en esta área. Especialmente porque muchos autores no divulgan su investigación en los medios consultados.

Debido a que los estudios que no contenían las palabras *Formal Verification* en el título no se incluyeron en el conjunto de estudios primarios, es posible que en el proceso de búsqueda se haya excluido un número significativo de estudios relacionados con el área de la investigación. Por otra parte, la inclusión de trabajos en talleres pudo alterar los resultados, debido a que su naturaleza es diferente respecto a la de las revistas y las conferencias. La dificultad de discernir los parámetros establecidos en la investigación para aquellas fuentes que solo permitían ver el *abstract* pudo haber influido en los resultados de la clasificación.

Con el fin de identificar las áreas de interés en cada uno de ellos, la Verificación Formal en los diferentes países y épocas se ha agrupado en áreas temáticas, lo que necesariamente no se corresponde con las establecidas para responder a las preguntas de investigación de esta investigación, sin embargo, de la misma revisión de la literatura surge la sugerencia de que diferentes funciones se asocian a diferentes necesidades y características de motivación. Al agrupar todos estos roles y funciones se pudo haber perdido parte del detalle que fue posible incluir en los análisis.

En esta revisión el término *Verificación Formal* engloba una multitud de roles en la Ingeniería del Software, como las tareas que llevan a cabo todos los profesionales que participan directamente en la producción de software. Esto genera limitaciones al estudio, porque rara vez se definen o diferencian individualmente de acuerdo a la práctica, pero también es cierto que las habilidades, roles y prácticas en esta área han cambiado durante la línea de tiempo cubierta por la revisión. Por ejemplo, a comienzos del 2000 el rol de programador/analista todavía era común, mientras que para mediados de 2010 se referenciaban como ingenieros de software. Por lo tanto, las investigaciones y las publicaciones relacionadas con la Verificación Formal también se pueden haber sesgado con estas corrientes.

4. CONCLUSIONES

El objetivo de este trabajo fue realizar una síntesis del estado del arte acerca de la investigación científica en la el área de la Verificación Formal, y para lograrlo se realizó una revisión sistemática de la literatura, considerada como el primer paso del paradigma de investigación basado en la evidencia.

En los últimos años la Verificación Formal se ha convertido en un medio práctico para detectar la presencia de comportamientos no deseados en los productos software, una propiedad requerida para los modelos críticos. Los modelos para comprobar la calidad en la industria del software y los utilizados por los probadores de teoremas avanzados, facilitan la realización de análisis complejos de las especificaciones de forma automática o semiautomática.

Por la naturaleza de la Verificación Formal el enfoque de investigación con mayor representatividad es el empírico, en parte por la necesidad de comprobar en un estudio de casos el modelo creado a través de la observación y el análisis de resultados.

Los artículos de investigación incluidos en este estudio abordan una amplia variedad de temas relacionados con la Verificación Formal, tales como las Redes Petri para dispositivos de control, circuitos digitales y procesadores (en los que se utilizan para realizar procesos de verificación exhaustiva para optimizar el diseño); la lógica temporal para verificar formalmente la concurrencia de acceso a los algoritmos de control y las especificaciones de seguridad de los sistemas de información para garantizar su seguridad; la semántica formal para las especificaciones del negocio; la verificación de los requisitos del sistema; el análisis de procesadores jerárquicos, los cuales se descomponen en un conjunto de condiciones para lograr una verificación más sencilla de razonar, permitiendo realizar la prueba en los diferentes niveles de arquitectura; y las heurísticas para verificar formalmente y automáticamente sistemas complejos como las próximas generaciones de microprocesadores.

La Ingeniería del Software se enfrenta a un reto permanente con la Verificación Formal, porque su objetivo es disminuir la brecha entre los sistemas de alta complejidad y la aplicabilidad de las buenas prácticas en todo el proceso de desarrollo.

La especificación formal es un tema que se detecta en todos los trabajos de investigación del estudio. Algunos describen la necesidad de establecer métodos de presentación y de redacción de especificaciones con características como: accesibilidad para el usuario basada en la representación lógica funcional del conocimiento, posibilidad de análisis automatizado de conversión y traducción a otros lenguajes desarrollados en modelos formales, el formato formal unificado para el intercambio entre diferentes sistemas de desarrollo y la representación gráfica de la lógica de las frases del lenguaje de programación.

Otra característica encontrada en los estudios primarios es que la Verificación Formal se integra en diferentes áreas a través de *frameworks*, que permiten el desarrollo de aplicaciones para verificar formalmente los sistemas que son independientes de la técnica de prueba subyacente y de las nuevas técnicas de verificación sobre el nivel de palabra, como la abstracción de predicados y la teoría del módulo de satisfacción.

Las preguntas de investigación planteadas en la metodología se respondieron de acuerdo con los resultados obtenidos en la revisión. Estos resultados se pueden utilizar en la industria y la academia para proyectar nuevas investigaciones y trabajos conducentes a la automatización de la Verificación Formal. Esta área es prioritaria para la comunidad, porque la complejidad de los sistemas de las décadas siguientes seguirá en incremento y la prueba manual no será suficiente.

Los resultados de esta revisión plantean nuevas preguntas que se podrían resolver en futuras investigaciones. Por ejemplo, debido a que los ingenieros de software han conformado un grupo profesional nuevo a los establecidos a finales de siglo en las Ciencias Computacionales, queda temáticas y cuestiones relacionadas con la Verificación Formal que siguen sin resolver, lo que genera la necesidad de estudios adicionales.

También sería útil examinar cómo vincular activamente a los métodos formales en los planes estudio de las Ciencias Computacionales, esto podría ofrecer como resultados futuros que la automatización total de las pruebas del software sea una realidad. Además, es necesario seguir trabajando para desarrollar un modelo matemático para *formalizar* la Ingeniería del Software.

REFERENCIAS

[1] Sülflow A. et al. (2009). WoLFram - A Word Level Framework for Formal Verification. En International Symposium on Rapid System Prototyping. Washington, USA.

[2] Copty F. et al. (2001). Efficient debugging in a formal verification environment. Lecture Notes in Computer Science 2144, 275-292.

[3] Glass R. et al. (2002). Research in Software Engineering: An analysis of the literature, Information and Software Technology 44(8), 491-506.

[4] Dyba T. y Dingsoyr T. (2008). Empirical studies of agile software development: A systematic review. Information and Software Technology 50(9-10), 833-859.

[5] Wohlin C. et al. (2000). Experimentation in Software Engineering: An introduction. Springer.

[6] Dyba T. et al. (2005). Evidence based software engineering for practitioners. IEEE Software 22(1), 58-65.

[7] Kitchenham, B. et al. (2004). Evidence based software engineering. En 26th International Conference on Software Engineering. Scotland, UK.

[8] Brereton P. et al: Lessons from applying the systematic literature review process within the software engineering domain. Journal of Systems and Software 80(4), 571-583.

[9] Kitchenham B. (2009). Procedures for Undertaking Systematic Literature Reviews. Joint Technical Report. Computer Science Department. Keele University.

[10] Kitchenham B. et al. (2009). Systematic literature reviews in software engineering: A systematic literature review. Journal Information and Software Technology 51(1), 7-15.

[11] Dyba T. y Dingsoyr T. (2008). Empirical studies of agile software development: A systematic review. Journal Information and Software Technology 50(9-10), 833-859.

SEGUNDA PARTE

INGENIERÍA DE REQUISITOS

CAPÍTULO XII
Significado y gestión del conocimiento en la Ingeniería de Requisitos[1]

Edgar Serna M.[1]
Oscar Bachiller S.[2]
Alexei Serna A.[1]
[1]Instituto Antioqueño de Investigación
[2]Universidad de Cundinamarca

Tradicionalmente se asume que la especificación de requisitos, como producto de la Ingeniería de Requisitos, tiene alto impacto en las demás fases del desarrollo de software. Por lo tanto, la gestión del conocimiento para constituirla se debería realizar de manera estructurada para descubrir, analizar y comprender la cadena datos-información-conocimiento, explícita y tácita, que posee las partes interesadas. En este capítulo se presenta los resultados de una revisión de la literatura orientada a responder: 1) ¿Qué significado tiene el conocimiento en la Ingeniería de Requisitos? 2) ¿Qué enfoques se proponen para gestionar el conocimiento en la Ingeniería de Requisitos? y 3) ¿Se puede evidenciar la eficiencia y la eficacia de los modelos para gestionar el conocimiento en la Ingeniería de Requisitos? La conclusión del análisis es que: 1) el conocimiento tiene un alto significado en esta fase, pero los autores todavía no se ponen de acuerdo sobre cuál sería la mejor manera de otorgarlo y aplicarlo; 2) no se encontró un marco general para establecer un enfoque validado para gestionar el conocimiento en la Ingeniería de Requisitos; y 3) las valoraciones a la eficiencia y eficacia de los modelos son bajos y, mayoritariamente, interpretaciones personales.

[1] Publicado en inglés en International Journal of Information Management 37, 155–161. 2017.

INTRODUCCIÓN

De forma generalizada, en la comunidad del desarrollo de software se acepta que la Ingeniería de Requisitos es la fase del ciclo de vida con mayor incidencia en la calidad del producto final. Pero, debido a la complejidad de los problemas actuales y a que todavía se aplica modelos tradicionales para gestionar el conocimiento que en ella se genera, es difícil alcanzar una comprensión rápida y objetiva de las necesidades de las partes interesadas. Para aportar en la búsqueda de soluciones, diversos autores trabajan desde hace tiempo en propuestas para gestionar ese conocimiento, pero, hasta el momento, no han encontrado una de amplia aceptación y reconocimiento en la comunidad.

Si la finalidad es aportar al mejoramiento de la calidad del software y para que los procesos formativos las atienda e incluya en sus contenidos curriculares, es necesario conocer el estado actual de estas investigaciones, lo mismo que de la eficiencia y eficacia de las propuestas. Dado que el objetivo a corto plazo de la comunidad del software es mejorar la calidad y, por tanto, la fiabilidad y seguridad de sus productos, se requiere que los ingenieros de software se capaciten en nuevas propuestas para gestionar el conocimiento en la Ingeniería de Requisitos [1].

Este problema no ha pasado desapercibido para los investigadores que, inclusive, se han motivado a realizar análisis sobre cómo gestionar el conocimiento en la Ingeniería de Requisitos. Jurisica et al. [2] (1999) afirman que la gestión del conocimiento en esta fase se ocupa de su representación, organización, adquisición, creación, uso y evolución en sus múltiples formas. Pero que se necesita mejorar la comprensión de cómo lo utilizan los individuos, los grupos y las organizaciones. Aunque su propuesta es interesante y fue validada, tiene una cobertura muy amplia, por lo que adaptarla para analizar la gestión del conocimiento en esta fase requiere mucho trabajo.

Bresciani et al. [3] analizan un marco de la gestión del conocimiento en la Ingeniería de Requisitos basado en agentes, con el objetivo de diseñar soportes para capturar y formalizar el incorporado o extraído de la organización. Es interesante observar cómo aplican su propuesta y validan sus resultados, aunque su inconveniente es que se basa en agentes, un principio de las Ciencias Computacionales que todavía se está abriendo paso en la investigación.

Andreas Breiter [4] adapta algunos modelos existentes al contexto de la Ingeniería de Requisitos para la gestión del conocimiento y, sobre esa base, deriva los requisitos funcionales específicos y los integra en el desarrollo del sistema a través de un proceso de diseño participativo. La propuesta de este autor se adapta fácilmente para gestionar el conocimiento, pero los formatos que propone no tienen el alcance suficiente. Andrade et al. [5] proponen introducir un programa de gestión del conocimiento que apoye el proceso del software, estructurado bajo un esquema de formalización y capaz de representar, capturar y transmitir el conocimiento que se pueda aprovechar en la Ingeniería de Requisitos. Su trabajo es uno de los pocos que demuestran cómo gestionar el conocimiento en esta fase, aunque su objetivo se orienta realmente a la Ingeniería del Software en general.

El trabajo de Al-Karaghouli et al. [6] tiene como objetivo construir un marco teórico orientado a cerrar las brechas entre los diferentes tipos de conocimiento, lo mismo que a gestionar los requisitos de negocio y los flujos de información entre las partes interesadas. Su propuesta es un marco práctico que describe algunas técnicas y herramientas derivadas, pero le falta demostrar su funcionamiento más allá de las áreas específicas sobre la cuales se sustenta, específicamente en la Ingeniería de Requisitos. Dominik Schmitz [7] orienta su trabajo a proporcionar medios

mejorados para soportar el conocimiento en la elicitación, el análisis, la documentación, y otras operaciones sobre los requisitos. Además, aborda la dinámica del proceso de la Ingeniería de Requisitos teniendo en cuenta su volatilidad. Aunque es novedosa, todavía queda por demostrar si es posible adaptarla al paradigma de la Programación Orientada por Objetos.

Chikh [8] afirma que en la Ingeniería de Requisitos se debe facilitar la colaboración entre las partes interesadas y los analistas, para que la gestión del conocimiento se minimice y se alcance mejores resultados. Su propuesta es un marco de gestión basado en el modelo SECI [9] de creación de conocimiento, cuyo objetivo es explotar el tácito y el explícito en relación con los requisitos dentro de un proyecto. El inconveniente de esta propuesta es que se restringe al modelo SECI, que no es lo suficientemente flexible para ajustarlo a contextos como la Ingeniería de Requisitos. Por su parte, Linda Schneider y sus colegas [10] afirman que en el desarrollo de software los requisitos no se identifican o implementan correctamente, porque el proceso depende, en gran medida, del conocimiento humano (explícito y tácito). Para resolver este problema identificaron los métodos asociados con la teoría de creación de conocimiento organizacional de Nonaka [11], y analizaron en qué medida ayudan a superar los problemas asociados. Aunque no es obvio ni fácil aplicarlos a los proyectos software, los métodos que identificaron proporcionan una aproximación para reducir el riesgo de la gestión del conocimiento.

En este trabajo se presenta los resultados de una revisión de la literatura para determinar el significado del conocimiento y cómo se gestiona en la Ingeniería de Requisitos. El objetivo es responder a las preguntas de investigación, a la vez que determinar si existe alguna manera de adaptar esas propuestas para gestionar este conocimiento, o si, por el contrario, es necesario estructurar un modelo de gestión del conocimiento diferente para la Ingeniería de Requisitos.

1. MÉTODO

De acuerdo con Brereton et al. [12] una revisión de la literatura se descompone en tres fases principales: 1) planificación, 2) realización, y 3) documentación, combinadas con otros procedimientos resumidos en seis actividades [13, 14]:

1. *Preguntas de investigación*. Para esta revisión se formularon tres preguntas: 1) ¿Qué significado tiene el conocimiento en la Ingeniería de Requisitos? 2) ¿Qué enfoques se proponen para gestionar el conocimiento en la Ingeniería de Requisitos? Y 3) ¿Se puede evidenciar la eficiencia y la eficacia de los modelos para gestionar el conocimiento en la Ingeniería de Requisitos?

2. *Proceso de búsqueda*. El objetivo inicial en esta investigación fue identificar estudios candidatos, para lo cual se diseñó un plan de consulta a las bases de datos ACM, IEEE, Sciendirect, Springer y Willey. Los parámetros de búsqueda incluyeron palabras clave como: *Knowledge management*, *Requirements Engineering*, *models*, *methodologies*, *knowledge types*, *meaning of knowledge*, que debían estar contenidas por lo menos una vez en el documento.

3. *Criterios de inclusión y exclusión*. El principal criterio de inclusión fue la relevancia del trabajo para responder las preguntas de investigación, por lo cual se tuvo en cuenta criterios tales como: ser investigación explicita, estar en la línea de tiempo establecida, presentar una descripción teórica, describir una aplicación práctica, detallar un estudio de caso, presentar un modelo o una metodología para gestionar conocimiento, y reconocer el trabajo de otros. Inicialmente, el aporte se descarta si no cumple por lo menos con uno de estos criterios.

4. *Valoración de la calidad*. Para determinar la calidad de los aportes se tuvo en cuenta criterios tales como: formalidad y pertinencia del medio de difusión, autoridad del autor, calidad de los resultados y las fuentes de datos, nivel de sustentación de la tesis, proceso de investigación aplicado, coherencia entre resultados y conclusiones, aceptabilidad (citaciones que ha recibido), valoración del aporte por parte de la comunidad y reconocimiento de la industria luego de experimentar la propuesta. A cada uno se asignó un criterio de valor.

5. *Recopilación de datos*: Se creó una matriz con: 1) tipo: artículo, capítulo de libro, libro, presentación en evento, otro; 2) título; 3) autor; 4) aporte: descripción teórica, aplicación práctica, caso de estudio, modelo, metodología; y 5) año. Se encontraron 83 documentos.

6. *Definir el análisis de datos*. En esta fase se aplicó la propuesta de Dyba y Dingsoyr [15] para analizar una serie de documentos y filtrar el conjunto de estudios primarios: 1) identificar los estudios relevantes, 2) excluir estudios con base en el título, 3) excluir estudios con base en los resúmenes, y 4) analizar y seleccionar los que hacen un aporte relevante a la investigación con base en el texto completo. Teniendo en cuenta los criterios de inclusión-exclusión y de valoración de la calidad, en este análisis se extrajeron 29 trabajos de la muestra inicial. Posteriormente, se realizó el cruce de información para determinar la eficiencia y eficacia de cada aporte, entonces se descartaron otros 14 trabajos. Luego de esta fase, la muestra final quedó conformada por 40 documentos, cuyo análisis se presenta a continuación.

2. RESULTADOS

De la discusión y el análisis de los trabajos de la muestra final se obtuvo la información necesaria para responder las preguntas de investigación. Cada uno de los autores realizó de forma individual un primer acercamiento de análisis, y posteriormente se llevó a cabo una discusión conjunta para responder las preguntas formuladas.

2.1 Significado del conocimiento en la Ingeniería de Requisitos

Conocimiento es un término que tiene diferentes significados en diferentes contextos, por ejemplo, de acuerdo con el diccionario Webster, es el hecho o condición de *saber algo* con familiaridad, que se ha adquirido a través de la experiencia o la asociación. Para Davenport y Prusak [16] es una mezcla de experiencias, valores e información contextual y especializada que se originan y aplican en *la mente de cada conocedor*, y que proporcionan un marco para evaluar e incorporar nuevas experiencias e información. En opinión de Biggs y Tang [17], son ideas o entendimientos que poseen las personas y que utilizan para tomar decisiones eficaces, por lo tanto, *es específico e individual*. En el contexto organizacional es la suma de lo que se conoce y reside en la inteligencia y las capacidades *de las personas*.

De acuerdo con estas definiciones el conocimiento se origina y reside en las personas, por lo que se puede clasificar y caracterizar, tal como lo hacen Nonaka y Takeuchi [18], quienes denominan conocimiento *tácito* al que conforma las habilidades técnicas y las dimensiones cognitivas de las personas, y conocimiento *explícito* al que comunican, difunden y transmiten. Biggs y Tang [17] denominan los tipos de conocimiento como: declarativo-proposicional (*saber qué*), procedimental (*saber hacer*), condicional (*saber cómo*) y funcional (*saber cuándo*). En todo caso, las características del conocimiento es que debe ser válido, útil, claro, pertinente y tener significado e importancia en un contexto [19]. De acuerdo con estos autores, el conocimiento es personal, y para aprovecharlo hay que contextualizarlo. El contexto que se utilizará en este trabajo para analizar el significado del conocimiento y sus implicaciones es la *Ingeniería de Requisitos*.

En esta fase del desarrollo de software se intercambia permanentemente datos e información entre las partes interesadas y el equipo de analistas. La habilidad que debe desarrollar este equipo es saber cómo convertirlos en conocimiento, de tal manera que les permita adquirir la sabiduría suficiente para comprender, modelar y presentarle una solución al problema. En la etapa de la elicitación de requisitos se aplica diversas técnicas para recopilar las opiniones, apreciaciones, interpretaciones, datos e información que poseen los usuarios acerca del problema, al mismo tiempo que de las necesidades que debe satisfacer la solución. Uno de los problemas en este proceso es que la comunicación se realiza en lenguaje natural, lo que ocasiona incomprensiones, debido a su ambigüedad, lo que no facilita la obtención de información fiable para comprender el contexto y el domino de la situación a solucionar.

Para alcanzar una mejor comprensión de los problemas que atiende, los investigadores de la Ingeniería del Software han buscado maneras de aprovechar el conocimiento tácito y explicito que poseen las personas y las organizaciones. Aunque han encontrado inconvenientes, también han logrado casos de éxito en los que las técnicas funcionan; entonces, los equipos acumulan y aprovechan los datos y la información para crear el mapa y el modelo de la situación-problema. Esta situación se vive cotidianamente en la fase de la Ingeniería de Requisitos, porque muchas veces las partes interesadas interpretan que lo que saben no le aporta a la situación, o simplemente no quieren compartirlo. A continuación, se describe algunos trabajos en los que se analiza el significado del conocimiento en esta fase del ciclo de vida del software.

Lambe [20] distingue cuatro dimensiones desde las que se puede originar el conocimiento: 1) la información que posee cada persona, 2) lo que cada individuo ha adquirido con la experiencia y que puede transmitir, 3) lo que posee el grupo acerca del problema, y 4) la cultura y la historia de la organización. Con base en ellas Stephanie White [21] propone cómo aprovechar la información y el conocimiento de las partes interesadas en la Ingeniería de Requisitos: 1) capturar la información individual, 2) compartir lo aprendido, 3) resumir la información, y 4) integrar lo aprendido con la cultura e historia de la organización (enciclopedia del conocimiento). Lambe propone esta práctica como una manera de darle significado al conocimiento individual, para comprender las interpretaciones personales y delimitar y clarificar las ambigüedades, para luego modelar el problema y encontrarle una solución.

Abdulmajid [22] afirma que gran parte del conocimiento que se comparte en la Ingeniería de Requisitos se encuentra en documentos, y que las partes interesadas lo comprenden porque es *explícito*. Sin embargo, en las mentes de los participantes permanece un conocimiento sustancial e indocumentado, al que considera como *tácito*. Además, como no se ve y permanece inconsciente o conscientemente oculto, genera retos importantes al incluirlo en el que acumula el equipo. Su idea es crear debates para tratar cuestiones del problema que requieren aclaración, en las que cada participante sustente su posición con argumentos que solamente él conoce. Por lo tanto, debe recurrir a su conocimiento tácito para defenderla y, de esta manera, se aprovecha su experiencia, habilidades y fortalezas para darle significado al conocimiento en la elicitación.

Para Waqar Hussain [23], y de acuerdo con la filosofía de la investigación interpretativa, el conocimiento se adquiere solamente a través de construcciones sociales: lenguaje, conciencia, sentido común, documentos, herramientas y otros artefactos. En la Ingeniería de Requisitos esto significa reconocer las diferentes experiencias de los individuos acerca del problema, para integrarlas y analizarlas en profundidad y darles significado como requisitos a satisfacer. Para este autor el objetivo es superar el problema que genera la volatilidad de los requisitos, por lo que el equipo no puede permitir que primen las interpretaciones individuales, debido a que son subjetivas y se basan en conocimiento tácito.

Jiangping Wan y sus colegas [24] sostienen que el desarrollo de software es una empresa altamente dependiente del conocimiento, por lo que es fundamental darle significado antes de transferirlo. Para ellos, la clave reside en la forma como se captura el explícito y el tácito. En el primer caso, proponen combinar prácticas para tamizar los datos explícitos y delimitar el conocimiento sobre los procesos, luego aplicar prácticas de internalización en el equipo y así crear una interpretación del problema. En cuanto al tácito, sugieren aplicar prácticas de externalización, tales como diagramas y modelos causa-efecto, para que las partes interesadas modelen lo que saben del contexto del problema y le den significado.

De acuerdo con Markus Flückiger [25] desarrollar software es trabajar con conocimiento, especialmente porque en la Ingeniería de Requisitos la colaboración y la comunicación juegan un papel importante. Para este autor, el conocimiento para elicitar los requisitos proviene desde diferentes dimensiones: el modelo de negocio, las experiencias de las personas, las reglas del negocio y la tecnología involucrada. Por lo tanto, el equipo se debe esforzar por descubrirlo, integrarlo y darle significado en un lenguaje de interpretación común, mediante un proceso en el que debe primar el trabajo conjunto, además de evitar que las opiniones y posiciones personales oculten las conclusiones acordadas por todos.

Taheri y sus compañeros [26] afirman que elicitar requisitos implica trabajar con gran cantidad de conocimiento, pero que nos es fácil obtenerlo, utilizarlo y darle significado. Para ellos, ese conocimiento proviene de tres dimensiones: 1) la documentación, 2) las partes interesadas, y 3) el equipo de desarrollo, por lo que se crea categorías que exigen interpretación inmediata, antes que se conviertan en ruedas sueltas que dificulten la integración. Luego hay que definir algunos factores para medir el conocimiento y darle significado: completitud, correctitud y comprensibilidad. De esta manera el equipo modela el conocimiento adquirido y lo actualiza en la medida que descubra nuevo, o que se deseche el existente.

Para Mamoona Humayoun y Asad Masood Qazi [27] los equipos de Ingeniería de Requisitos necesitan modelar el conocimiento involucrado, porque, de otra manera, desperdician su uso potencial para comprender el problema. Su propuesta consiste en modelarlo a través de pasos interactivos: aprender, explorar, capturar, almacenar, compartir y explotar. Estos autores están convencidos que de esta manera se aprovecha y se da significado al conocimiento que proviene de las personas, la organización y la documentación.

2.2 Enfoques para gestionar el conocimiento en la Ingeniería de Requisitos

Para esta investigación se asume *enfoque* como una manera de ver las cosas o las ideas y de tratar los problemas relativos a ellas [28]. En los estudios analizados se identifica que los enfoques dependen significativamente del contexto particular de la organización y de la conformación del equipo que los pone en práctica. No fue posible determinar un marco general que represente una estructura claramente establecida sobre cómo se gestiona el conocimiento en la Ingeniería de Requisitos, porque en la revisión solamente se hallaron interpretaciones desde el punto de vista de los autores consultados. En general, se identificaron tres enfoques que engloban los puntos de vista centrados en la elicitación de requisitos, los cuales se tomaron como base para responder la segunda pregunta de esta investigación.

1. El primer enfoque corresponde a los *procesos de interacción social*, donde la adquisición y transferencia del conocimiento individual, a un conocimiento colectivo, adquiere relevancia característica, lo mismo que la manera cómo se organiza y comparte al interior de la organización. Este modelo se sustenta principalmente en la propuesta de Nonaka y Takeuchi

[18], quienes dan importancia particular al conocimiento formal e informal (tácito y explícito), centrado en la experiencia y en la no-ambigüedad. En la Ingeniería de Requisitos los analistas utilizan diferentes técnicas para elicitar las necesidades de las partes interesadas, pero, en el proceso, se presenta inconvenientes, porque no logran identificarlas y recolectarlas correctamente debido a las condiciones particulares de cada entorno, y porque los actores no comparten fácilmente todo su conocimiento sobre el problema [29, 30]. Además, debido a que los procesos de gestión y negociación se desarrollan en escenarios donde se presenta diversidad de intereses e interpretaciones, necesariamente el trabajo debe ser colaborativo [31], y la interacción social lo hace posible.

2. El segundo enfoque se relaciona con los *modelos basados en técnicas de Inteligencia Artificial*, orientados a asegurar la calidad del proceso de elicitación de requisitos. En este sentido se encontraron dos corrientes características:

   - *Sistemas de recomendación* [32], que se centran en: 1) recomendación de partes interesadas, identificando las personas que son capaces de brindar una descripción completa de los requisitos necesarios y de las más propensas a cooperar; y 2) recomendación de requisitos susceptibles de reutilización en una implementación.

   - *Principios de lógica difusa* [33], cuyo objetivo es lograr una mejor representación de la información, a partir de la automatización del proceso de generación de mapas conceptuales. De esta manera se brinda un mejor soporte para la interpretación y la toma de decisiones desde los datos que se recolectan en el desarrollo de proyectos. Esta estrategia se concibe como una buena herramienta para la gestión del conocimiento, porque los conceptos pueden ser capturados o consultados, de tal manera que permiten descubrir automáticamente las conexiones implícitas para generar nuevos mapas. Así se infiere nuevo conocimiento y, a partir del repositorio de conocimiento y de los registros históricos de la organización, se favorece la generación de una memoria organizacional.

3. El tercer enfoque se sustenta en métodos asociados con las *técnicas de dinámicas de juego* (*gamification*), potencializadas con el uso de metodologías y marcos de trabajo ágil para el desarrollo de software. Estos últimos se fundamentan en el hecho de que la elicitación no es un proceso trivial, porque en la interacción con las partes interesadas no se puede asegurar que se obtiene todos los requisitos, cuando simplemente se considera preguntas relacionadas con lo que se espera que haga el sistema a desarrollar [34]. También hay que tener en cuenta que, en la Ingeniería de Requisitos, el proceso de comunicación y la transferencia de conocimiento se pueden convertir en un problema, porque la diversidad cultural, temporal, geográfica y socio-económica de las partes interesadas es un obstáculo a superar [35]. Los autores consideran que técnicas tales como entrevistas, cuestionarios, casos de uso e historias de usuario, entre otras, son propensas a elicitar requisitos ambiguos o incorrectos [34]. Esta deficiencia conlleva a implementar características que, en muchas ocasiones, resultan innecesarias o erróneas, y a no considerar funcionalidades necesarias e importantes, por lo que son insuficientes para identificar todas las necesidades del sistema [35].

La conclusión en esta revisión de la literatura es que, si el objetivo es mejorar la calidad de la interacción entre las partes interesadas, se necesita promover nuevos enfoques para gestionar el conocimiento en la Ingeniería de Requisitos, en los que principios como las dinámicas de juego puedan brindar la realimentación necesaria y estimular la participación activa de los actores [34]. Esto podría incrementar la colaboración y la creatividad, generando una mejor transferencia del conocimiento ajustado a la naturaleza del proceso de elicitación, donde básicamente se identifica

ideas por medio del trabajo colaborativo [35]. En este proceso se obtiene consenso sobre lo que se va a desarrollar, mejorando la especificación de requisitos y su gestión y negociación, a la vez que se incrementa la calidad y la aceptación del software [36].

2.3 Eficiencia y eficacia de los modelos para gestionar el conocimiento en la Ingeniería de Requisitos

En la literatura existe diversas opiniones en relación con el significado del término *modelo* y de los indicadores de *eficiencia* y *eficacia* [37]. Para esta investigación se asume modelo como una abstracción teórica de una situación o problema, cuya finalidad es reducir su complejidad y realizar predicciones concretas sobre el mismo. En el caso de eficiencia y eficacia, aunque en muchos textos se asumen como sinónimos, aquí se tratan con significado propio y diferente: *eficiencia* es la medida del éxito que se logra al gestionar el conocimiento entrante versus la obtención de requisitos, es decir, es el rendimiento de la relación entre el conocimiento que entra y el conocimiento que sale en la Ingeniería de Requisitos [38]; mientras que *eficacia* se utiliza para conocer el grado en que los modelos logran encontrar, asimilar y difundir el conocimiento, necesario para comprender el problema en esta fase del ciclo de vida [39].

En los estudios analizados la eficiencia se toma como la capacidad de obtener el máximo rendimiento posible del conocimiento que se obtiene de las *fuentes origen*. En términos generales tratan de encontrar: 1) la factibilidad de obtener un mejor rendimiento con el conocimiento existente, y 2) la posibilidad de mejorar ese rendimiento a partir de la gestión del conocimiento. En relación con la eficacia, tratan de encontrar el nivel de logro de los objetivos establecidos en la Ingeniería de Requisitos a partir del conocimiento compartido. Otra cuestión que presentan los autores se refiere a los indicadores para medir la eficiencia y la eficacia de los modelos. Para ellos un buen indicador es: 1) *útil*, porque responde a las preguntas relacionadas con las metas y objetivos; 2) *confiable*, porque su uso produce siempre los mismos resultados a través del tiempo, en las mismas condiciones y con los mismos temas; 3) *válido*, porque refleja con precisión el concepto que se está utilizando para medir; 4) *oportuno*, porque debe estar disponible el tiempo suficiente para informar las decisiones; y 5) *rentable*, porque no debe ser excesivamente costoso.

Para responder la tercera pregunta de investigación se seleccionó los modelos divulgados que tuvieran algún tipo de análisis valorativo a su eficiencia y eficacia, los cuales se describen en la Tabla 1. Se aclara que los trabajos seleccionados también cumplen con los criterios de inclusión-exclusión y de valoración de la calidad, determinados para esta revisión de la literatura.

Tabla 1. Modelos para gestionar el conocimiento en la Ingeniería de Requisitos

Modelo	Descripción
Wiig Knowledge Management Cycle (WKMC) [40]	Su propósito es facilitar la creación, acumulación, despliegue y uso de conocimiento de calidad. En la Ingeniería de Requisitos esto significa que, si el objetivo es adquirir conocimiento relevante para la definición de requisitos, el equipo debe tener la experiencia para abordar las fases.
Espiral TIC para los procesos de gestión del conocimiento [41]	Este modelo utiliza TIC como ayuda en la gestión del conocimiento. El equipo toma el conocimiento tácito y explícito desde las diferentes dimensiones, para interiorizarlo, socializarlo, exteriorizarlo y combinarlo en la búsqueda de los requisitos.
Integrated Knowledge Management Systems (IKMS) [42, 43]	Se puede considerar como una extensión de KMS con modelos de software mejorados, módulos y retroalimentación, para operar en dominios de información más amplios. Proporcionan acceso a la cadena datos-información-conocimiento necesaria para elicitar y analizar requisitos. Además, facilita el acceso a la información pertinente para comprender el problema.

Knowledge Management Software Process Improvement (KMSPI) [44]	Se trata de un modelo teórico-relacional orientado a facilitar el descubrimiento y adquisición del conocimiento que las personas poseen sobre un problema. Posteriormente, y mediante tareas iterativas, el equipo lo asimila y comparte para identificar los requisitos.
Customer Knowledge Management (CKM) [45]	Se define como un proceso continuo de generar, difundir y utilizar el conocimiento de las partes interesadas al interior del equipo de trabajo, y entre éste y las partes interesadas. El objetivo es gestionar y aprovechar todo tipo de conocimiento que poseen acerca, sobre y para la comprensión del problema [46].
Knowledge Management Systems (KMS) [47]	Son Sistemas de Información para gestionar el conocimiento. En la Ingeniería de Requisitos su objetivo es recuperar, transferir y difundir el conocimiento, estructurado o no-estructurado, explícito o tácito, de las partes interesadas, para comprender y solucionar problemas y tomar decisiones [48].

Con base en estos hallazgos y de acuerdo con lo definido para responder la pregunta sobre la eficiencia y la eficacia de los modelos para gestionar el conocimiento en la Ingeniería de Requisitos, en la Tabla 2 se presenta el resumen de los indicadores encontrados y la valoración que los investigadores les otorgan en el funcionamiento de los modelos.

Tabla 2. Valoración de la eficiencia y eficacia de los modelos

Indicadores de gestión del conocimiento	Valoración					
	WKMC	Espiral TIC	IKMS	KMSPI	CKM	KMS
Crear/Descubrir	++		++	++	++	++
Entender	+	++	++	++	+	+
Comprender						
Compartir	++	++	+	++	++	++
Aplicar					++	+
Experimentar	++	++		+		
Validar						
Documentar	+	++	++	+		+
Actualizar						
Innovar						

3. ANÁLISIS DE RESULTADOS

En relación con los enfoques para gestionar el conocimiento en la Ingeniería de Requisitos, en esta revisión no fue posible encontrar un marco general que represente una estructura claramente establecida para hacerlo. Esto se debe a que los trabajos analizados reflejan claramente los puntos de vista de los autores y de sus experiencias en contextos específicos, pero los procesos en la Ingeniería de Requisitos no se pueden estandarizar para todas las situaciones. Los enfoques hallados en esta revisión de la literatura reflejan la necesidad de innovarlos, porque, de acuerdo con los mismos autores, todavía no es posible afirmar que alguno sea suficiente para gestionar el conocimiento en esta fase del ciclo de vida.

Aquí lo que se requiere es un enfoque interaccional estructurado, de manera que sus principios y axiomas le permitan al equipo encontrar el conocimiento y darle significado para su posterior gestión. Es decir, hay que entender que la Ingeniería de Requisitos es una práctica social de comunicación [49], en la que hay que tener en cuenta: 1) el principio de *totalidad*, porque el todo es más que la suma de sus partes, 2) el principio de *causalidad circular*, porque la elicitación de requisitos es un juego de implicaciones mutuas y de acciones y retroacciones circulares, y 3) el principio de *regulación*, porque los procesos en la Ingeniería de Requisitos deben seguir normas y conveniencias establecidas como reglas de juego. Esta interacción entre el equipo y las partes interesadas podría ser la herramienta que permita recopilar el conocimiento necesario para la comprensión del problema y la especificación de los requisitos.

Por otro lado, la eficiencia y la eficacia de los modelos para gestionar el conocimiento en la Ingeniería de Requisitos no recibe altas valoraciones por parte de los investigadores. De los seis modelos analizados en este trabajo se desprende que:

1. La máxima valoración otorgada es de Bueno (++) para algunos indicadores y de Aceptable (+) para otros.
2. Ninguno ha sido valorado para todos los indicadores de la gestión del conocimiento. De los 10 indicadores se encontró valores para cinco en tres de los modelos y para cuatro en los otros tres.
3. Los indicadores que reciben mejores valoraciones son Entender y Compartir, aunque con apreciaciones diversas. Esto se debe a que los modelos buscan, primero, entender el conocimiento, para luego compartirlo obviando la comprensión. De esta manera la gestión del conocimiento queda incompleta y los requisitos no se elicitan adecuadamente.
4. El modelo de Espiral TIC es el mejor valorado para los indicadores de Entender, Compartir, Experimentar y Documentar, pero no se encontraron valores para los otros seis.
5. Los modelos más CKM y KMS reciben las valoraciones más bajas. Esto se entiende porque la industria y la academia todavía pueden estar en proceso de comprensión y aplicación, y todavía no publican sus comentarios.

4. CONCLUSIONES

En los trabajos analizados en esta revisión se percibe la importancia del conocimiento en la Ingeniería de Requisitos. Los autores están de acuerdo en que la valoración a la cadena datos-información-conocimiento es esencial para comprender el problema y estructurar una solución. Para ellos, darle significado al conocimiento explícito y tácito de las partes interesadas, es el primer paso para elicitar sus necesidades adecuadamente, a la vez que se convierte en la base para que los equipos de trabajo puedan especificar los requisitos con claridad.

Pero, aunque reconocen su importancia, todavía no se ponen de acuerdo sobre cuál sería la mejor manera de darle significado al conocimiento que recopilan mediante las diferentes técnicas de elicitación de requisitos. Porque el explícito lo pueden encontrar con relativa facilidad, mientras que el tácito se convierte en visiones e interpretaciones personales, y muchas veces no hallan cómo integrarlo y darle un significado que complemente lo que han aprendido desde el explícito. Esta personalización crea una barrera para entender y comprender lo que realmente expresan las partes interesadas y para contextualizarlo en el problema. Algunas de las propuestas se dirigen específicamente a convencer a los actores para que comuniquen lo que saben del contexto y, aunque algunas pueden funcionar en determinadas situaciones, la mayoría fracasa, porque no tienen en cuenta que es necesario darle significado para asociarlo con el conocimiento explícito.

Con base en los resultados de esta investigación se puede afirmar que la gestión del conocimiento en la Ingeniería de Requisitos apenas es *Aceptable*. Entonces, si el objetivo es aportar para mejorar la fiabilidad y seguridad de los productos software y satisfacer las demandas de una sociedad software-dependiente, se necesita estructurar y promover nuevos modelos para gestionar el conocimiento en esta fase del ciclo de vida.

De esta manera, se valida la hipótesis de la investigación desde la que se desprende esta revisión de la literatura, en el sentido de que *la gestión del conocimiento en la Ingeniería de Requisitos no es adecuada para la complejidad de los problemas actuales*. Por lo tanto, se recomienda tomar

las mejores prácticas de los enfoques y modelos, propuestos hasta el momento, para innovar la forma en que se gestiona el conocimiento mediante un modelo integrado y validado.

REFERENCIAS

[1] Terstine M. (2015). The progress of requirements engineering research. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 5(1), 18–24.

[2] Jurisica I. et al. (1999). Using ontologies for knowledge management: An information systems perspective. En Annual Conference of the American Society for Information Science. San Francisco, USA.

[3] Bresciani P. et al. (2003). Requirements engineering for knowledge management in egovernment. En M. Wimmer (Ed.), Knowledge management in electronic government (pp. 48–59). Springer.

[4] Breiter A. (2004). Requirements development in loosely coupled systems: Buildinga knowledge management system with schools. En 37th Hawaii International conference on system sciences. Honolulu, USA.

[5] Andrade J. et al. (2006). A reference model for knowledge management insoftware engineering. Engineering Letters 13(2), 159–162.

[6] Al-Karaghouli et al. (2008). Knowledge management: Usinga knowledge requirements framework to enhance UK health sector supplychains. En European and Mediterranean conference on information systems. Madrid, Spain.

[7] Schmitz D. (2010). Managing dynamic requirements knowledge – An agent-basedapproach. PhD. Dissertation. RWTH Aachen University.

[8] Chikh A. (2011). A knowledge management framework in software requirements engineering based on the SECI model. Journal of Software Engineering andApplications 4, 718–728.

[9] Gourlay S. (2003). The SECI model of knowledge creation: Some empiricalshortcomings. En 4th European conference on knowledgemanagement. Oxford, UK.

[10] Schneider L., et al. (2013). Knowledge creation in requirements engineering −Asystematic literature review. En 11th international conference on wirtschaftsinformatik. Leipzig, Germany.

[11] Nonaka I. (1994). A dynamic theory of organizational knowledge creation.Organization Science 5(1), 14–37.

[12] Brereton P. et al. (2007). Lessons from applying the systematic literature review process within the software engineering domain. Journal of Systems and Software 80(4), 571–583.

[13] Kitchenham B. (2003). Procedures for undertaking systematic literature reviews. Jointtechnical report. Keele University.

[14] Kitchenham B. et al. (2009). Systematic literature reviews in software engineering: A systematic literatura review. Information and Software Technology 51(1), 7–15.

[15] Dyba T. y Dingsoyr T. (2008). Empirical studies of agile software development: A systematic review. Information and Software Technology 50(9–10), 833–859.

[16] Davenport T. y Prusak L. (1998). Working knowledge: How organizations manage what they know. Harvard Business School Press.

[17] Biggs J. y Tang C. (2011). Teaching for quality learning at university. McGraw-Hill.

[18] Nonaka I. y Takeuchi H. (1995). La organización creadora de conocimiento. Cómo las compañías japonesas crean la dinámica de la innovación. Oxford University Press.

[19] Denning S. (2000). The springboard: How storytelling ignites action in knowledge-era organizations. Butterworth Heinemann.

[20] Lambe, P. (2007). Organizing knowledge and organizational effectiveness. Chandos Publishing Ltd.

[21] White S. (2010). Application of cognitive theories and knowledge management to requirements engineering. En 4th annual IEEE systems conference. San Diego, USA.

[22] Abdulmajid M. (2010). Facilitating tacit-knowledge acquisition within requirements engineering. En 10th WSEAS international conference on applied computer science. Iwate, Japan.

[23] Hussain W. (2010). Requirements change management in global software development: A case study in Pakistan. Master Tesis. Linnaeus University.

[24] Wan J. et al. (2011). Research on explicit and tacit knowledge interaction in software process improvement project. Journal of Software Engineering and Applications 4(6), 335–344.

[25] Flückiger M. (2011). The brain's perspective on requirements engineering. En fourth international workshop on managing requirements knowledge. Trento, Italy.
[26] Taheri L. et al. (2014). Identifying knowledge components in software requirement elicitation. En IEEE international conference onindustrial engineering and engineering management. Las Vegas, USA.
[27] Humayoun M. y Qazi A. (2015). Towards knowledge management in RE –Practices to support software development. Journal of Software Engineering and Applications, 8(8), 407–418.
[28] Bunge M. y Ardila R. (2002). Filosofía de la psicología. Siglo XXI Editores.
[29] Burnay C. et al. (2014). What stakeholders will or will not say: A theoretical and empirical study of topic importance in requirements engineering elicitation interviews. Information Systems 46, 61–81.
[30] Vásquez D. et al. (2014). Knowledge Management acquisition improvement by using software engineering elicitation techniques. Computers in Human Behavior 30, 721–730.
[31] Azedegan A. et al. (2013). Applying collaborative process design to user requirements elicitation: A case study. Computers in Industry 64(7), 798–812.
[32] Felfernig A. et al. (2013). An overview of recommender systems in requirements engineering. En W. Maalej y G. Thurimella (Eds.), Managing requirements knowledge (pp.315–332). Springer.
[33] Wang W. et al. (2008). Self-associated concept mapping for representation, elicitation and inference of knowledge. Knowledge-BasedSystems 21(1), 52–61.
[34] Ribeiro C. et al. (2014). Gamifying requirement elicitation: Practical implications and outcomes in improving stakeholder's collaboration. Entertainment Computing 5(4), 335–345.
[35] Ghanbari H. et al. (2015). Utilizing online serious games to facilitate distributed requirements elicitation. Journal of Systems and Software109, 32–49.
[36] Snijders R. et al. (2015). REfine: A gamified platform for participatory requirements engineering. En 1st international workshop on crowd-based requirements engineering. Ottawa, Canada.
[37] Mouzas S. (2006). Efficiency versus effectiveness in business networks. Journal of Business Research 59(10–11), 1124–1132.
[38] Low J. (2000). The value creation index. Journal of Intellectual Capital 1(3), 252–262.
[39] Zheng W. et al. (2010). Linking organizational culture, structure, strategy, and organizational effectiveness: Mediating role of knowledge management. Journal of Bus. Research 63(7), 763–771.
[40] Dalkir K. (2005). Knowledge management in theory and practice. Elsevier.
[41] Pérez D. y Dressler M. (2007). Tecnologías de la información para la gestión del conocimiento. Intangible Capital 15(3), 31–59.
[42] Botha A. et al. (2008). Coping with continuous change in the business environment, knowledge management and knowledge management technology. Chandice Publishing Ltd.
[43] Zaki T. et al. (2008). Integrated knowledge management system (IKMS). En IEEE conference on technologies for homeland security. Orlando, USA.
[44] O'Connor R. y Basri S. (2011). Knowledge Management in software process improvement: A case study of very small entities. En M. Ramachandran (Ed.), Knowledge engineering for software development life cycles: Support technologiesand applications (pp. 273–288). Information Science Reference.
[45] Buchnowska D. (2011). Customer knowledge management models. Assessment and proposal. Lecture Notes in Business Information Processing 33, 25–38.
[46] Buchnowska D. (2014). Social CRM for customer knowledge management. Contemporary Economy 5(4), 65–80.
[47] Ghorab K. y Hegazy F. (2014). The influence of knowledge management on organizational business processes' and employees' benefits. International Journal of Business and Social Science 5(1), 148–172.
[48] Turban E. et al. (2011). Decision support and business intelligence systems. Pearson.
[49] Kirkwood S. et al. (2016). Towards an interactional approach to reflective practice in social work? European Journal of Social Work 19(3–4), 1–17.

CAPÍTULO XIII
Madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

Desde finales del siglo XX se ha publicado una serie de propuestas para Gestionar el Conocimiento en la Ingeniería de Requisitos que, si bien proceden del concepto de gestión tradicional basado en el mando y el control, en el futuro deberán dar lugar a un nuevo concepto de gestión que se ocupe más de desarrollar, apoyar, conectar, aprovechar y empoderar al equipo como trabajadores del conocimiento. En este capítulo se presenta los resultados de una investigación con el objetivo de determinar el nivel de madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos. Se realizó una revisión de la literatura acerca de las propuestas para gestionar este conocimiento y de las valoraciones que publican los investigadores luego de experimentarlas. Se seleccionaron siete propuestas, se encontraron ocho características valoradas y seis niveles de valoración, y se estructuró un modelo de madurez de cuatro niveles: 0. Infantil, 1. Adolescente, 2. Adulto, y 3. Veterano. La conclusión es que, hasta el momento, el nivel de madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos es Adolescente.

[1] Publicado en la Revista Ibérica de Sistemas e Tecnologias de Informação E17, 123-141. 2019.

INTRODUCCIÓN

En toda organización se crea documentos, se digita datos en las aplicaciones, se envía información mediante correos u oficios impresos y de muchas otras maneras, donde se documenta los procesos diarios de trabajo y producción. Por otro lado, los empleados intercambian ideas diariamente, para informar o clarificar dudas o discutir temas formales o informales sobre sus funciones. Todo esto es un proceso permanente en el que se refina los datos y la información hasta convertirla en conocimiento [1, 2]. El problema con ese conocimiento es que, mayoritariamente, reside en los individuos y, esporádicamente, en pequeños grupos. De esta manera se puede catalogar como explícito o tácito de acuerdo con la forma como se comparte: si se deja en documentos o se publica por algún medio, entonces se considera *explícito*, porque es posible utilizarlo y aplicarlo en las actividades donde se requiera; pero si solamente se encuentra en la mente de los empleados, por experiencia o porque simplemente no lo desean compartir, entonces es *tácito*, porque está oculto y no se publica ni se comunica.

En este sentido, el objetivo de la Gestión del Conocimiento es lograr un uso satisfactorio y productivo para el modelo organizacional o el objetivo del proceso [2]. Por eso es que, para la Ingeniería de Requisitos, es tan importante gestionar adecuadamente el conocimiento de los actores involucrados en esta fase del ciclo de vida del software. A este respecto es conveniente tener en cuenta que toda la gestión parte de la cadena datos-información-conocimiento relacionada con el problema que se desea resolver, y que se rige por las fases del conocimiento organizacional: 1) *creación*, cuando se comparte, formal o informalmente, datos o información entre los actores (a partir de dimensiones internas o externas al problema); 2) *captura*, que se logra cuando los actores lo documentan o almacenan en el cuerpo de conocimiento del problema; 3) *transformación*, cuando el equipo de trabajo lo organiza, mapea y convierte de forma transdisciplinar en una herramienta de solución al problema; 4) *capitalización*, cuando el equipo lo almacena en la Enciclopedia de Conocimiento de la empresa y se convierte en activo de valor para resolver el problema; y 5) *aprovechamiento*, el equipo utiliza el conocimiento capitalizado para construir el documento de la especificación de requisitos.

Aunque en la Ingeniería del Software cada problema es único y cada modelo-solución tiene una metodología específica, esos modelos tienen en común una serie de procesos complejos orientados a encontrarle una solución al problema. Además, encontrar una solución software consiste de aplicar un enfoque sistemático, disciplinado y cuantificable para al desarrollo, operación y mantenimiento del producto. Por lo que los integrantes del equipo que lleva a cabo la Ingeniería de Requisitos se consideran trabajadores del conocimiento [3], y deben ser capaces de crear, capturar, transformar y capitalizar el conocimiento, explícito y tácito, que poseen las partes interesadas acerca del problema. De otra manera no podrán elicitar adecuadamente los requisitos, que el producto-solución debe satisfacer con calidad, fiabilidad y seguridad [4]. Pero este es un tema que todavía necesita investigación, porque, prácticamente, ninguno de los modelos de desarrollo de software tiene explícitamente incorporada una propuesta para gestionar el conocimiento en la Ingeniería de Requisitos [5].

En la práctica, la Ingeniería de Requisitos se desarrolla en un ambiente de conocimiento intensivo, por lo que el equipo de trabajo lo debe gestionar para aprovecharlo eficientemente en pro de encontrar una posible solución al problema [5, 6]. En la Figura 1 se presenta la proximidad entre las fases de la Gestión del conocimiento y las etapas de la Ingeniería de Requisitos, donde el objetivo es encontrar el conocimiento explícito y transformar el táctico, para generar un documento de especificación de requisitos sólido y fiable. En esta fase del ciclo de vida el conocimiento es diverso y creciente, por lo que los equipos tienen problemas para identificarlo,

ubicarlo y encontrar la mejor manera de utilizarlo [2]. Esto se debe a que primero hay que caracterizarlo y determinar las dimensiones de origen: en los procesos de negocio, la toma de decisiones, lo empresarial, lo declarativo, lo procedimental y la heurística, y luego darle el nivel de formalidad o informalidad necesario [7].

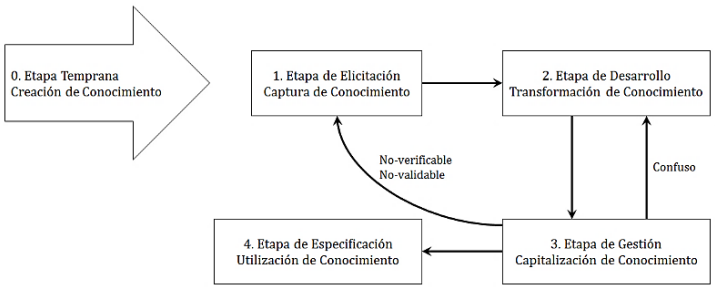

Figura 1. Proximidad entre las fases de la Gestión del Conocimiento y las etapas de la Ingeniería de Requisitos

Pero materializar esta proximidad entre las fases y las etapas no es una tarea trivial, porque quien lo posee debe *traducirlo* a un lenguaje natural y utilizar canales de comunicación verbales y no-verbales para transferirlo a los demás integrantes del equipo, para que juntos lo decodifiquen de acuerdo con sus modelos mentales [8]. Otro factor que incrementa la complejidad de esta actividad es el hecho de que el equipo y las partes interesadas son profesionales de diversas disciplinas (transdisciplinares), con experiencias, perspectivas, intereses y expectativas diferentes, que tienen datos-información-conocimiento sobre el problema, tanto tácitos como explícitos, y casi siempre producto de su familiaridad, relación y función en el mismo. Esto genera lo que se conoce como *simetría de la ignorancia*, es decir, una brecha de conocimiento entre lo que creen saber del problema y lo que realmente saben del mismo, lo que dificulta el proceso de comunicación [8].

Diversos autores han presentado propuestas para gestionar el conocimiento en la Ingeniería de Requisitos, sin embargo, aplicarlas aisladamente, tal como se hace actualmente, no es suficiente [9]. Si el objetivo es logar una gestión eficiente, primero hay que analizarlas y determinar su nivel de madurez y eficiencia, para encontrar lo mejor de cada una y seleccionar la que mejor se adapte a las generalidades de la Ingeniería de Requisitos. La idea es que sean altamente efectivas, que minimicen el tiempo entre las fases de la Gestión del Conocimiento, y que permitan la conversión de lo tácito en explícito de manera ágil. Por lo tanto, este trabajo se sustenta en la hipótesis de que las propuestas, para gestionar el conocimiento en la Ingeniería de Requisitos, no tienen un adecuado nivel de madurez para atender la complejidad de los problemas actuales, pero que, al integrarlas, se podrían utilizar como punto de partida para desarrollar un modelo de Gestión del Conocimiento para Ingeniería de Requisitos.

1. MARCO REFERENCIAL

En términos generales, el objetivo de la Gestión del Conocimiento es mejorar la manera como se comparte el conocimiento en las organizaciones y gestionar sus flujos. Nonaka y Takeuchi [10] propusieron el modelo Socializar, Externalizar, Combinar e Interiorizar SECI, para convertir y aprovechar el conocimiento en las organizaciones, porque el conocimiento se crea a partir de las interacciones sociales. En la *Socialización* el conocimiento se transfiere entre individuos mediante el intercambio de modelos mentales y habilidades técnicas; en la *Externalización* el tácito se convierte en explícito mediante el desarrollo de modelos, protocolos y guías; en la *Combinación* la recombinación o reconfiguración de los cuerpos de conocimiento existentes crea nuevo conocimiento; y en la *Internalización* se alcanza el aprendizaje mediante procesos repetitivos.

El interés por gestionar el conocimiento adecuadamente en la Ingeniería de Requisitos ha llevado a diversos autores a emprender investigaciones para encontrar la manera de hacerlo. En sus propuestas tratan la cuestión del conocimiento en las etapas de la Ingeniería de Requisitos, tales como la *elicitación* [11], el *desarrollo* [12] o la *especificación* [13], a la vez que desarrollan propuestas metodológicas [11], herramientas específicas [14] o técnicas [15], y Gacitua et al. [16] investigan el papel del conocimiento en la Ingeniería de Requisitos y proponen cómo mitigar y gestionar sus efectos; sin embargo, ninguno enfrenta el problema desde una perspectiva de la Gestión del Conocimiento.

Por otro lado, algunos autores han adoptado el modelo SECI para aplicarlo en la Ingeniería de Requisitos, tales como Pilat y Kaindl [17], quienes proponen una perspectiva de Gestión del Conocimiento para crear conocimiento; Wan et al. [18] plantean una propuesta para convertir el conocimiento en las actividades de la elicitación de requisitos, con el objetivo de minimizar la simetría de la ignorancia entre los integrantes del equipo de trabajo. Por su parte, Vázquez et al. [19] realizan una clasificación de las técnicas de elicitación y proponen cómo seleccionarlas de acuerdo con el modelo SECI, su objetivo es reducir las dificultades de una adecuada selección. Todos los autores mencionados hasta el momento comparten una perspectiva de Gestión del Conocimiento, pero ninguno formula una propuesta, metodología, estrategia o proceso para determinar el nivel de madurez de las implementaciones en Ingeniería de Requisitos.

Por otro lado, pero centrados en la Ingeniería del Software, Olmos y Ordas [20] describen un proceso basado en conocimiento para la elicitación de requisitos. Para ellos todavía persiste muchos problemas en la industria del software, porque utiliza técnicas informales y no se dispone de algoritmos definidos que puedan ayudar a la solución. Esta falta de madurez en las técnicas aplicadas crea ambigüedades e inconsistencias, por lo que lo más conveniente es gestionar el conocimiento de otra manera. Khalid y sus colegas [21] proponen un modelo de conocimiento para ayudar a mejorar la eficiencia de las propuestas de Gestión del Conocimiento, utilizando un repositorio central de conocimientos como servidor en la nube. La propuesta de Ward y Aurum [6] se orienta a desarrollar un modelo de conocimiento, que se ha aplicado en algunas organizaciones de software para modelar el conocimiento. Los autores hacen una descripción de la inmadurez de otras propuestas para gestionar el conocimiento en la Ingeniería del Software.

Por su parte, Kess y Haapasalo [22] argumentan que, esencialmente, la Ingeniería del Software es un proceso en el que la Gestión del Conocimiento se realiza marginalmente, por lo que estructuran una propuesta para integrarlo. Dingsøyr y sus colegas [23] describen los problemas de la industria por falta de propuestas adecuadas para la gestión del conocimiento, mientras que Aurum et al. [24] reafirman la necesidad de buenas propuestas para hacerlo en la Ingeniería del Software, para lo que proponen la solución como un reto para los investigadores y detallan algunos factores clave que pueden tener en cuenta para lograrlo. En este sentido, Li [25] aporta un procedimiento para compartir y crear conocimiento en empresas de software; Wongthongtham y Chang [26] describen las ontologías como modelos para gestionar el conocimiento; Georgiades et al. [27] definen una solución basada en la semántica del lenguaje natural; y Cuena y Molina's [28] identifican como una falacia la implementación de la Gestión del Conocimiento en las fases de la Ingeniería de Software tradicional.

Otro asunto que no ha permitido una adecuada maduración de las propuestas para gestionar el conocimiento en la Ingeniería de Requisitos, es que no se encuentra unanimidad en el lenguaje utilizado para formular las terminologías y estrategias de Gestión del Conocimiento [29]. Estos autores afirman que el problema central de las propuestas es que los actores no se comunican adecuadamente, debido a que tienen culturas y profesiones diferentes. White [30] afirma que el

equipo de trabajo debe ser capaz de comprender las acciones humanas y las normas culturales de cliente, en relación con el problema. Debido a que el proceso de la Ingeniería de Requisitos es básicamente trabajo en equipo, el problema debe estar bien definido, utilizar prototipos, un vocabulario común para la comunicación, un ambiente de apoyo y abordar la gestión del conocimiento en forma de anotaciones y símbolos para resolverlos por separado [31]. Para estos autores, aquí es donde radica la falta de eficiencia de las propuestas para hacerlo. Además, deben tener en cuenta que el conocimiento, como los requisitos, no proviene de una fuente específica, sino que se origina en diversas dimensiones [32], tales como el dominio, el contexto, el medio ambiente, la experiencia, el grupo y la cultura.

También se encontraron algunos esfuerzos relacionados con la madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos: algunos centrados en herramientas TIC [33]; otros sugieren metodologías para medirla en partes específicas de la Gestión del Conocimiento [34] o en nichos específicos de la Ingeniería de Requisitos [35]; mientras que Gacitua y sus colegas [16] intentan hacerlo de forma más amplia, y discuten indirectamente cómo afecta el nivel de madurez de las propuestas actuales a la especificación de requisitos.

2. MÉTODO

Por la naturaleza de esta investigación se utilizó el Diseño de Investigación Científica DSR, que es fundamentalmente una metodología para resolver problemas con el objetivo de crear innovaciones que definan ideas, prácticas, capacidades técnicas o productos, a los que se puede mejorar su eficiencia y efectividad mediante el análisis, el diseño, la implementación, la gestión y el uso de las experiencias publicadas [36]. Además de las actividades en las diferentes etapas de la Ingeniería de Requisitos, DSR tiene su ciclo de investigación, por lo que es una metodología ampliamente utilizada en la realización de investigaciones relacionadas con el desarrollo de software. Esta metodología se estructura alrededor de varias directrices y, para esta investigación, se toman las siguientes: 1) relevancia del problema, 2) contribuciones al cuerpo del conocimiento, 3) rigor de la investigación, 4) diseño como proceso de búsqueda, y 5) comunicación de resultados [37]. La base del conocimiento puede ser teorías, métodos, experiencias, especialistas, procesos y artefactos, pero el rigor de la misma depende de una adecuada selección, aplicación y evaluación de ese conocimiento para analizarlo y obtener los resultados previstos.

Una actividad que pretenda ser eficiente y objetiva, como es el caso de las involucradas en la Ingeniería del Software, se convierte en un tema relevante para un equipo de desarrollo de software, porque debe dedicar hasta un 60% del tiempo del ciclo de vida a tratar de entender su entorno, a través de una adecuada gestión del conocimiento en la Ingeniería de Requisitos. Por eso es que este problema es relevante para la comunidad. DSR es una metodología inherentemente iterativa, por lo que el diseño como proceso de búsqueda se basa en una revisión de la literatura, a la que se añade la experiencia personal de los autores y de los especialistas consultados. Por otro lado, es esencialmente un proceso de búsqueda para descubrir el estado de la cuestión y proponer una solución al problema.

Para esta investigación se adapta la propuesta DSR de Hevner [37] de tres ciclos, como se observa en la Figura 2. En el *ciclo de diseño* se diseña, analiza y evalúa los procesos de búsqueda que se aplicarán; en el *ciclo de relevancia* se itera la búsqueda en las bases de datos hasta encontrar los trabajos relevantes para responder a las preguntas de investigación; y en *el ciclo de rigor* se construye el cuerpo de conocimiento con base en la validación de las valoraciones a las propuestas analizadas.

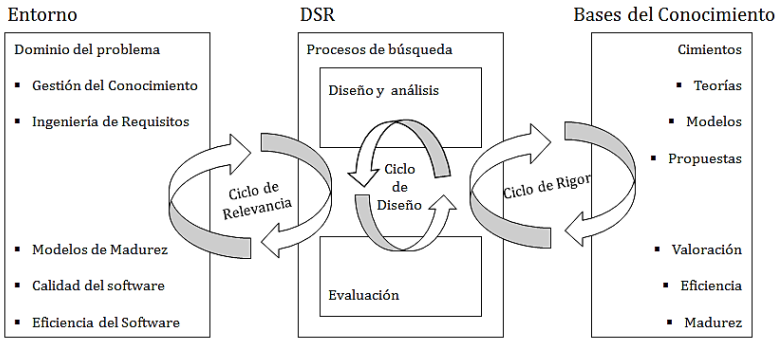

Figura 2. Método de investigación (adaptado de [37])

3. RESULTADOS

Las preguntas en las que se basa esta investigación son:

1. ¿Qué Propuestas para Gestionar el Conocimiento en la Ingeniería de Requisitos PGCIR se han publicado?
2. De acuerdo con la evaluación a estas propuestas, ¿cuál es el nivel de madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos?

3.1 PGCIR encontradas

Luego de aplicar el *ciclo de diseño* de la metodología y llevar a cabo una búsqueda de las características del entorno de investigación, se encontraron 24 PGCIR. En el *ciclo de relevancia* se realiza una evaluación concreta, en la que se tuvo en cuenta la estructura, la relevancia del autor en este campo y el seguimiento que ha recibido cada propuesta con base en las citaciones recibidas. Posteriormente, en el *ciclo de rigor* se extracta las validaciones y valoraciones para cada propuesta, para construir las bases de conocimiento que orientan la selección de la muestra. Al final de este proceso se seleccionaron las *siete* propuestas que se describen en la Tabla 1.

Tabla 1. PGCIR seleccionadas en la revisión de la literatura

Propuesta	Descripción	Observaciones
ISO STEP [38]	Se puede utilizar para intercambiar datos e información en varias industrias, entre ellas la del software, especialmente en la Ingeniería de Requisitos.	▪ Proporciona información específica del dominio, y los protocolos ofrecen conocimiento específico [39]. ▪ Se trata de un riguroso intento por identificar y demostrar una estrategia de razonamiento en la Ingeniería de Requisitos [40]. ▪ Los requisitos se expresan en lenguaje natural, con los problemas de lo tácito [41]. ▪ Se adapta en el tiempo para clarificar y actualizar los requisitos. No resuelve el problema de ambigüedad acerca de qué es un requisito y los actores no logran una comprensión clara del problema [42].
MTKISDP [43]	Se basa en los principios de comprender, conocer y elicitar. Da prioridad al conocimiento crítico necesario y define la base subyacente para su implementación en lenguaje formal.	▪ Presenta un modelo general para la elicitación, se centra en la selección de la técnica y deja la gestión del conocimiento en un segundo plano [44]. ▪ No logra capturar el conocimiento relevante para cada dominio porque la Gestión del Conocimiento debe abarcar mucho más que identificar y elicitar [45]. ▪ Permite representar los usuarios y utiliza grupos y comunidades; no tiene en cuenta las dimensiones ni la transdisciplinariedad de los requisitos [46].

		▪ Otros autores han publicado análisis indirectos a esta propuesta [47-51]; no logra una valoración sobresaliente para las características que se proponen en esta investigación.
RMKMSE [52]	Se compone de un esquema de formalización para representar, capturar y transmitir el conocimiento relevante. Se centra en el mantenimiento del software.	▪ Es una experiencia de uso en Gestión del Conocimiento no-exitosa [53]. ▪ Se orienta esencialmente a la captura del conocimiento en la Ingeniería de Requisitos; usa la memoria corporativa como principal fuente; no aprovecha eficientemente a las partes interesadas [54]. ▪ Se focaliza en la participación en el conocimiento y la reutilización de las lecciones aprendidas; deja de lado aspectos importantes de la Ingeniería de Requisitos [55, 56]. ▪ No aprovecha suficientemente el conocimiento de los actores involucrados; se centra básicamente en la organización [57].
KBRE [58]	Es un modelo integrado de sistemas expertos y modelado específico de dominio. Uno define las reglas y propiedades del sistema y el otro requiere una herramienta metacase.	▪ Es un método centrado en cómo modelar y utilizar eficazmente el conocimiento del dominio, pero se olvida del tácito que poseen las partes interesadas; satisface algunas características de madurez, pero deja de lado las más necesarias [67].
KMPRE [58]	Se basa en el conocimiento compartido acerca del problema y su dominio. Su objetivo es superar el problema del intercambio de conocimiento en la Ingeniería de Requisitos, enfatizando en la importancia de elicitar a partir del mismo.	▪ Es un enfoque en el que predomina la comunicación entre las partes; facilita la transferencia y el intercambio de conocimiento [59]. ▪ Es incompleta, costosa y requiere tiempo; no logra la transformación ni la transferencia de conocimiento en la dimensión que se requiere [60]. ▪ Se basa en el modelo SECI [10] y ofrece ideas y técnicas básicas para comprender y facilitar el proceso de transferencia de conocimiento y transformación, pero nada más [61].
KMFSE-SECI [62]	Utiliza ontologías para describir la especificación y representar el contenido de los requisitos; se sustenta en el modelo SECI.	▪ Necesita ir más allá del conocimiento explícito, porque en la Ingeniería de Requisitos el tácito es muy importante [63]. ▪ Es algo flexible y adaptable, pero es difícil de aplicar a otras funcionalidades [64].
KMoS-RE [60]	Se orienta a la transformación y transferencia de conocimiento; incorpora métodos para identificar, capturar, indexar y hacer explícito el tácito.	▪ Tiene en cuenta el modelo de Jackson [65] y se basa en SECI [10]. ▪ Es un modelo de conocimiento sobre una estrategia para Ingeniería de Requisitos; no tiene en cuenta la complejidad y la dimensionalidad, el dominio informal estructurado, ni las ambigüedades [66].

3.2 Madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos

Para determinar la madurez se estructuró el siguiente procedimiento: 1) identificar y definir las características valoradas por los autores, 2) seleccionar los niveles de valoración aplicados, 3) resumir los niveles de valoración a cada característica con base en los aportes de los autores y sus análisis, 4) construir un modelo de madurez para analizar las valoraciones, y 5) encontrar el nivel de madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos.

1. *Características de valoración.* En el contexto de la investigación, y a partir de la estructura y definición de sus etapas e interrelaciones, se construyó un marco de evaluación al nivel de madurez de las PCGIR con base en las descripciones encontradas en la revisión de la literatura. El objetivo fue estructurar un marco de referencia y análisis para identificar su eficiencia y

eficacia, teniendo en cuenta los diferentes aspectos que evaluaron los autores en la experimentación de las propuestas. En esta investigación se encontró que los investigadores tienen en cuenta que la propuesta debe ser sistémica, además de responder a las características propias de los sistemas actuales, como se resumen en la Tabla 2.

Tabla 2. Características de valoración

Característica	Descripción
Flexibilidad F	Se puede implementar en todo tipo de problemas, sin importar tamaño, grado de complejidad o nivel de exigencia.
Adaptabilidad A	Se puede adecuar en función del contexto y tipo de organización, el equipo de trabajo, la infraestructura, los objetivos estratégicos y la base de conocimiento, entre otros.
Multidimensionalidad Mu	Permite la selección de diversas estrategias para gestionar el conocimiento proveniente de las dimensiones involucradas en el problema.
Transdisciplinariedad T	Permite gestionar la cadena datos-información-conocimiento proveniente de una amplia variedad de disciplinas.
Evolutividad E	Facilita el desarrollo de la Gestión del Conocimiento en la medida que las etapas de la Ingeniería de Requisitos evolucionan hacia el documento de especificación, y que la comunicación crecen en alcance y complejidad.
Realimentación R	Se regula a sí mismo y cada resultado de un sub-proceso incide en el todo, reintegrándolo y modificándolo, permitiendo que el modelo tenga capacidad de control y autocorrección.
Medible Mi	Ofrece métricas para los resultados, la evolución y el impacto de la gestión del conocimiento en el logro de la especificación.
Prospectiva P	Permite identificar oportunidades de mejora y de realizar modificaciones o cambios oportunos en la gestión y líneas de acción adoptadas por el equipo.

2. *Niveles de valoración*. En la Tabla 3 se describe los niveles de valoración tomados de los mismos análisis que realizan los autores a las PGCIR.

Tabla 3. Niveles de valoración

Valor	Detalle	Descripción
0	Inexistente	No lo permite (Nunca)
1	Inicial	La permite parcialmente, es adecuada y corresponde a lo esperado (Casi nunca)
2	Repetible	Está homologada, pero no está estandarizada ni tiene gobernabilidad (Rara vez)
3	Definida	Está homologada, no está estandarizada, pero tiene gobernabilidad (A veces)
4	Administrada	Está completamente homologada en la funcionalidad del modelo, está estandarizada para las distintas etapas y cuenta con gobernabilidad (Casi siempre)
5	Optimizada	Opera completamente y se puede refinar hasta niveles de mejores prácticas, se basa en los resultados de mejoras y diseños continuos (Siempre)

3. *Valoración de las características*. En la Tabla 4 se presenta la valoración a las características de las PGCIR, resultado del análisis hecho por los autores consultados en la experimentación de las mismas.

Tabla 4. Valoración de las características de las PGCIR

Propuesta	Características valoradas							
	F	A	Mu	T	E	R	Mi	P
ISO STEP - AP233	3	3	0	0	2	2	3	0
MTKISDP	1	1	0	0	3	2	3	0
RMKMSE	2	2	0	0	2	2	2	0
KBRE	2	1	0	0	0	3	2	1
KMPRE	1	2	0	0	0	2	2	1
KMFSE-SECI	3	3	0	1	1	2	1	1
KMoS-RE	3	3	1	0	1	3	3	1

4. *Modelo de madurez aplicado*. Conocer el nivel de madurez de Gestión del Conocimiento en la Ingeniería de Requisitos será posible en la medida que las propuestas para hacerlo demuestren eficacia y eficiencia, y esto se logra ubicando el proceso dentro de un nivel determinado. Aunque la mayoría de investigadores están familiarizados con los niveles de madurez utilizados por CMM, en esta investigación se propone determinar esta madurez aplicando una perspectiva con niveles comparativos al proceso en los seres humanos, es decir: Infantil, Adolescente, Adulto y Veterano.

   - *Nivel 0: Infantil*: cuando necesita mucho cuidado, atención y *afecto*; sus características son:

     - La gestión del conocimiento se realiza solamente al finalizar la fase, porque no está en sintonía con el ciclo de vida del desarrollo de software.
     - Los problemas de comunicación hacen que la gestión falle y que el equipo no llegue a un acuerdo dialogado, porque la falta de un lenguaje común invalida cualquier intento de conclusión.
     - Se genera re-procesos porque, en cualquier caso, hay que modificar o actualizar la cadena datos-información-conocimiento que se intenta gestionar.
     - El equipo de trabajo invierte más tiempo en tratar de concluir, que en comprender la Transdisciplinariedad y Multidimensionalidad de la cadena que desea gestionar.
     - La organización que presenta el problema impone la visión de una disciplina desde la que se debe realizar la gestión del conocimiento.
     - Debido a que hay que *cuidar*, *mimar* y prestarle mucha *atención*, la gestión del conocimiento no se puede dejar sin vigilancia por mucho tiempo, por lo que se puede afirmar que el nivel de madurez de este proceso desalienta, en lugar de alentar al equipo a seguir adelante.

   - *Nivel 1: Adolescente*: cuando se puede aplicar y seguir sin mucho detalle durante un tiempo razonable, pero todavía se desconfía de su *responsabilidad*; sus características son:

     - Un error en la gestión ocasiona problemas en el resto de las etapas de la propuesta.
     - Es importante conocer el error, pero no se necesitará mucho esfuerzo para demostrarlo.
     - Se desperdicia mucho tiempo intentando analizar la causa de cada problema a la vez que se deja de ejecutar varias etapas de la propuesta.
     - Para encontrar los problemas, el equipo debe solucionar las inconsistencias y volver a correr las etapas recorridas, un ciclo que se tiene que repetir muchas veces.
     - En este nivel, y al igual que con los humanos adolescentes, la gestión del conocimiento es sorprendentemente útil, pero se comporta de manera irresponsable y puede causar más daño que beneficio a la Ingeniería de Requisitos.

   - *Nivel 2: Adulto*: cuando es *digna* de confianza y se puede aplicar sin mucha *vigilancia* durante algunas etapas de la Ingeniería de Requisitos, pero todavía se desvía de las *responsabilidades*; sus características son:

     - Al final de cada etapa el equipo logra analizar buena parte de la cadena datos-información-conocimiento.
     - Aunque el equipo todavía no llega a conclusiones dialogadas, los aportes disciplinares se tienen en cuenta y se analiza cómo integrarlos.
     - La gestión del conocimiento es algo más que reunir datos-información-conocimiento.
     - El equipo se puede concentrar en encontrar y corregir los problemas de comunicación, mientras progresa en las demás etapas de la propuesta sin intervención.

- En este nivel la gestión del conocimiento no requiere vigilancia extrema y, aunque el proceso es responsable y se puede confiar en él, todavía no es auto-dirigido, porque su madurez no ha llegado a un nivel de independencia que le permita al equipo dejar que la propuesta se guie autónomamente.

- *Nivel 3: Veterano*: cuando ha evolucionado y crecido a través de los niveles y es posible dejar que siga su curso sin *vigilancia permanente*; el conocimiento es totalmente gestionable; es autónoma; se analiza, discute y concluye la cadena datos-información-conocimiento con aportes desde todas las disciplinas y dimensiones necesarias. Sus características son:

  - La gestión del conocimiento es eficiente y eficaz, y el equipo desarrolla todo el proceso de forma transdisciplinar y multidimensional.
  - El equipo reutiliza la cadena datos-información-conocimiento en sus conclusiones, porque la entiende y dialoga permanentemente.
  - Al final de la Ingeniería de Requisitos se tiene conocimiento validado y maduro, y una serie de reportes que denotan la calidad y fiabilidad de la gestión realizada.
  - La Gestión del Conocimiento se estructura a partir de una propuesta planificada, que involucra la Transdisciplinariedad y la Multidimensionalidad de la cadena datos-información-conocimiento.
  - Se puede aplicar procedimientos de planeación estratégica en todas las etapas.
  - La gestión del conocimiento es un proceso autónomo en cuanto a reproducción y selección de la cadena de entrada, y a la validación, discusión y exposición de resultados.
  - La organización cuenta con un modelo de gestión del conocimiento reutilizable y fácil de proyectar a cada solución analizada.
  - Como en el caso de los humanos, en este nivel de Gestión del Conocimiento aporta experiencia y madurez, con el objetivo de que el producto de esta fase del ciclo de vida sea de utilidad para las demás.

5. *Madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos*. Con base en los resultados presentados en la Tabla 4 y en las opiniones, reflexiones y críticas que hacen los investigadores a las propuestas, en la Tabla 5 se presenta el nivel de madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos, calculado a partir de los resultados de la evaluación a las características que hacen los autores consultados, los niveles de valoración definidos y el modelo de madurez aplicado.

Tabla 5. Madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos

Características	Valoración promedio	Nivel de madurez			
		Infantil	Adolescente	Adulto	Veterano
Flexibilidad (F)	Definida		X		
Adaptabilidad (A)	Definida		X		
Multidimensionalidad (Mu)	Inicial	X			
Transdisciplina (T)	Inicial	X			
Evolutividad (E)	Inicial	X			
Capacidad de realimentación (R)	Repetible		X		
Medible (Mi)	Repetible		X		
Prospectiva (P)	Inicial	X			

4. ANÁLISIS DE RESULTADOS

Al analizar la aplicación de las diferentes propuestas en la industria, el equipo de investigadores se encontró con evaluaciones que desprecian cada una cada de ellas, pero al consultar con los

equipos de Ingeniería de Requisitos se pudo constatar que las empresas hacen prevalecer los costos y el tiempo en el ciclo de desarrollo, por lo que ellos están presionados a no seguir las fases estructuradas, sino a utilizar las que puedan necesitar. Debido a esta realidad, en la que pocas empresas tienen la paciencia y los recursos para aplicar el proceso completo de las propuestas, los autores toman la decisión de analizar únicamente los comentarios y evaluaciones que otros investigadores han publicado, luego de experimentar con las PGCIR.

Por otro lado, el problema se complica, porque los clientes se han vuelto cada vez más exigentes y los sistemas software cada vez más complejos. El aseguramiento de la calidad se está convirtiendo en una condición permanente para la sobrevivencia de las empresas y, actualmente, es una realidad en la industria del software. A pesar de que en ciertas esferas se afirma que la calidad del software ha mejorado en los últimos años, todavía está muy lejos del escenario ideal y de la madurez esperada, esencialmente porque la Ingeniería de Requisitos todavía presenta falencias al especificar las necesidades que debe satisfacer el producto. Para colaborar a llenar este vacío muchos investigadores han estructurado y publicado diferentes propuestas para gestionar el conocimiento en esta fase, pero, de acuerdo con los resultados analizados en esta investigación, todavía son limitadas y muchas no se centran directamente en la Gestión del Conocimiento.

Aunque la Ingeniería de Requisitos es una fase esencial para desarrollar un producto que satisfaga al cliente, y una parte integral de todo el ciclo de vida del desarrollo de software, la Gestión del Conocimiento en ella requiere un nivel de madurez que la haga eficiente, efectiva y flexible. Estas características son cada vez más complicadas de lograr, por lo que continuamente se incrementa la demanda por una Ingeniería de Requisitos madura, que asegure que la especificación de requisitos ocurra sobre una base regular de Gestión del Conocimiento. Además, que facilite la construcción de escenarios de comprensión del problema cada vez más reales, que requieran menos esfuerzo y que les permita a los equipos de desarrollo obtener y gestionar el conocimiento tácito y explícito sobre el sistema en desarrollo. Pero, de acuerdo con los resultados de esta investigación, los autores que han experimentado las PGCIR afirman que todavía no se puede demostrar que lo logran.

De acuerdo con los resultados obtenidos en esta investigación algunas empresas utilizan las PGCIR como una especie de *bala de plata*, que podría resolver problemas en la calidad del producto, ayudar a elicitar y especificar todos los requisitos y ahorrar tiempo y esfuerzo. Este no es el caso, porque la implementación implica una curva de aprendizaje progresivo, hasta que el equipo desarrolle las habilidades necesarias. Eso significa que, en realidad, en los primeros proyectos la propuesta seleccionada puede incrementar el esfuerzo y el costo de la Ingeniería de Requisitos. También es importante destacar que no se encontró una propuesta genérica que se pudiera aplicar en los diversos contextos y problemas en los que se desarrolla el software. Esto significa que la mayoría requiere alguna adaptación previa y la reinducción del equipo en cada caso, porque no hay algo así como una propuesta de *talla única*.

La creencia de que una propuesta determinada les ayudará a reducir costos y tiempos de entrega, hace que las empresas se convenzan de poder cumplir con los plazos. Pero es poco probable que alguna de ellas ayude en este sentido, porque los resultados demuestran que no reducen inmediatamente el esfuerzo y el tiempo, debido a que requieren capacitación y el desarrollo de habilidades para utilizar de forma dinámica. Además, la complejidad del software no da cabida a una reutilización inmediata de ninguna herramienta de Gestión del Conocimiento, y las empresas piensan en términos de fiabilidad y reutilización, en lugar de simplemente en la eficacia de la gestión. Otro asunto que la industria no comprende a cabalidad es que, actualmente, no todo el

conocimiento en Ingeniería de Requisitos se puede descubrir, comprender y gestionar, y muchos usuarios y clientes *esconden* conceptos clave, lo que representa un problema, porque es poco probable que una PGCIR por sí sola pueda ser capaz de encontrar el cien por ciento de ese conocimiento.

En términos generales, y de acuerdo con los resultados en esta investigación, se puede concluir que el nivel de madurez de la Gestión del Conocimiento en la Ingeniería de Requisitos, como área de investigación, desarrollo y aplicación, es *Adolescente*. Una conclusión a la que llegan los autores luego de analizar e interpretar los datos y valoraciones que los investigadores publican, luego de experimentar con las propuestas seleccionadas en la revisión de la literatura. Mejorar este nivel es una estrategia necesaria y, para muchas empresas, la única opción para optimizar los estándares de calidad del software, pero en esta investigación se encontró que todavía se halla en el proceso de maduración que requiere cualquier aplicación compleja en el mundo actual.

Por otro lado, en la práctica las características evaluadas en cada una de las propuestas se encuentran en diferentes niveles de madurez, porque en cada etapa de la Ingeniería de Requisitos surge un conocimiento único, que le permite al equipo pasar a la gestión en la etapa siguiente. Pero un problema para lograrlo es que actualmente en la industria esta gestión se observa como un proceso más del desarrollo de software, y las propuestas para hacerlo se estructuran, al parecer, sin una planeación real de las necesidades de la industria. Esto genera algunos desafíos que debe afrontar una PGCIR para alcanzar un nivel de madurez *Veterano*:

- El tiempo que se invierte para aplicarla no debe extender las actividades de la Ingeniería Requisitos.
- Los cambios permanentes en la formulación y los escenarios de los requisitos le exigen adaptación continua.
- Debe gestionar el conocimiento que generan tanto los requisitos explícitos como los tácitos.
- La industria invierte dinero, tiempo y esfuerzo en el desarrollo de software, por lo tanto, la propuesta debe permitir alcanzar resultados rápidamente.
- Las personas con las habilidades para gestionar conocimiento eficientemente son escasas, por lo que la propuesta debe ser fácil de comprender y aplicar.
- Un desafío importante para que una PGCIR se utilice y mejore su nivel de madurez, como área de investigación y de desarrollo, es el costo de aplicación. La industria opta por no aplicar herramientas que incrementen el tiempo o el valor del desarrollo de software.

5. CONCLUSIONES

Es importante observar que en las últimas décadas se ha incrementado el interés de los investigadores por divulgar Propuestas para Gestionar el Conocimiento en la Ingeniería de Requisitos PGCIR, en parte, motivados por aportar al mejoramiento y la escalabilidad del proceso de desarrollo de software. Al mismo tiempo, también llama la atención que, aunque se conozca y comparta estas propuestas, a la industria parece que le faltara algo: *no las aplican rigurosamente*.

Una de las principales cosas que recalcan sus autores es la importancia de no saltarse las fases, porque para que sea eficiente y eficaz para el equipo de trabajo, la PGCIR seleccionada se debe aplicar en su totalidad. De acuerdo con el modelo de madurez que se propone en este trabajo, las personas *crecen* a través de cada etapa y a partir de lo que lograron en la anterior, es decir, no es posible dejar de ser niño para pasar automáticamente a ser adulto, aunque se intente

desesperadamente, porque si se salta la adolescencia esa persona está destinada a fracasar en la vida. Lo mismo sucede con las PGCIR.

El cambio de paradigma y el aprovechamiento de la Gestión del Conocimiento en todo tipo de actividades está creciendo rápidamente, en parte porque las organizaciones se han dado cuenta de su potencialidad y porque los investigadores han innovado y mejorado sus propuestas. Esto ha permitido que se pueda utilizar en áreas como la Ingeniería de Requisitos para buscar una mejor especificación. Los conceptos y procesos fundamentales que se analizan en esta investigación reflejan un cambio creciente en el pensamiento y la práctica de la Gestión del Conocimiento en el desarrollo de software. Este cambio de paradigma, aunque procede del concepto de gestión tradicional, basado en el mando y el control, debe dar lugar a un nuevo concepto de gestión que se ocupe más de desarrollar, apoyar, conectar, aprovechar y empoderar a los integrantes del equipo como trabajadores del conocimiento.

La Gestión del Conocimiento les interesa a muchos profesionales e investigadores en el entorno de la Ingeniería del Software, y muchas organizaciones la prueban como un potencial repositorio de soluciones relacionadas con problemas en el desarrollo de software. Los desarrollos recientes en TI permiten compartir el conocimiento, documentado o no, independientemente del tiempo y el espacio. En este sentido, se puede prever que en el futuro se podrá capturar desde, y difundir en, diversos formatos y dimensiones, permitiéndole al equipo de trabajo elicitar los requisitos a cualquier escala y complejidad. Pero esto depende en gran medida de la eficiencia y efectividad de las PGCIR y del mejoramiento en el nivel de madurez de esta área de investigación y desarrollo.

Teniendo en cuenta que uno de los factores clave del éxito de un proyecto software reside en el hecho de que su propósito responde al cumplimiento de un objetivo estratégico de la organización, la Gestión del Conocimiento en la Ingeniería de Requisitos debe responder a sinergias de valor agregado y a dinámicas de intercambio en la cadena datos-información-conocimiento, lo que supone un importante reto socio-cultural-administrativo para el equipo de trabajo. Por eso es que una propuesta para gestionar el conocimiento que se comunica, analiza y difunde en esta fase, debe ser dinámica y adaptativa al problema y al propósito de la organización, a la vez que evolutiva conforme se asimila o ajusta el conocimiento tácito y explicito que se elicita.

REFERENCIAS

[1] Davenport T. (2010). Process Management for Knowledge Work. Handbook on Business Process Management. Springer.

[2] Rus I. y Lindvall M. (2002). Knowledge Management in Software Engineering. IEEE Software 19(3), 26-38.

[3] Ibarra Y. y Valle R. (2017). Knowledge management - A way to contribute to risk decision making in video game development. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software, 7(1), 68-73.

[4] Camacho J. et al. (2013). Understanding the Process of Knowledge Transfer in Software Engineering: A Systematic Literature Review. Int. Jour. of Soft. Comp. and Soft. Eng. 3(3), 219-229.

[5] Natali A. y Falbo R. (2002). Knowledge Management in Software Engineering Environments. En XVI Brazilian Symposium on Software Engineering. Vitoria, Brazil.

[6] Ward J. y Aurum A. (2004). Knowledge Management in Software Engineering - Describing the Process. En Australian Software Engineering Conference. Melbourne, Australia.

[7] Vasconcelos J. et al. (2009). A Knowledge-Engine Architecture for a Competence Management Information System. En UK Academy for Information Systems Conference. Oxford, UK.

[8] Achive T. (2018). Coexistence of the natural and the artificial in the mind: Engineering the non-possible. Actas de Ingeniería 4(1), 35-42.

[9] Shadbolt N. y Smart P. (2015). Knowledge elicitation. En J. Wilson y S. Sharples (Eds.), Evaluation of human work (pp. 163-196). CRC Press.
[10] Nonaka I. y Takeuchi H. (1995). The knowledge-creating company: How Japanese companies foster creativity and innovation for competitive advantage. Harvard Business Review 69(6), 96-104.
[11] Friedrich W. y Van der Poll J. (2007). Towards a methodology to elicit tacit domain knowledge from users. Interdisciplinary Journal of Information, Knowledge, and Management 2, 178–193.
[12] Medeni T. et al. (2011). Tacit knowledge extraction for software requirements specification: A proposal of research methodology design and execution for knowledge visualization. En 55th Annual Meeting of the ISSS. Hull, UK.
[13] Stone A. y Sawyer P. (2006). Identifying tacit knowledge-based requirements. IEEE Proceedings-Software 153(6), 211–218.
[14] Stoiber R. y Glinz M. (2009). Modeling and managing tacit product line requirements knowledge. En Second Intern. Workshop on Managing Requirements Knowledge. Atlanta, USA.
[15] Mohamed A. (2010). Facilitating tacit-knowledge acquisition within requirements engineering. En 10th International Conference on Applied Computer Science. Iwate, Japan.
[16] Gacitua R. et al. (2009). Making Tacit Requirements Explicit. En Second International Workshop on Managing Requirements Knowledge. Washington, USA.
[17] Pilat L. y Kaindl H. (2011). A Knowledge Management Perspective of Requirements Engineering. En Fifth International Conference on Research Challenges in Information Science. Gosier, Guadeloupe.
[18] Wan J. et al. (2010). Research on knowledge creation in software requirement development. Journal of Software Engineering and Applications 3(5), 487-494.
[19] Vázquez D. et al. (2012). Combining software engineering elicitation technique with the knowledge management lifecycle. International Journal of Knowledge Society Research 3(1), 1-13.
[20] Olmos K. y Ordas J. (2013). A Strategy to Requirements Engineering Based on Knowledge Management. Requirements Engineering 19, 421-440.
[21] Khalid S. et al. (2015). The Role of Knowledge Management in Global Software Engineering. En International Conference on Industrial Engineering and Operations Management. Dubai, Dubai.
[22] Kess P. y Haapasalo H. (2002). Knowledge Creation through a Project Review Process in Software Production. International Journal of Production Economics 80(1), 49-55.
[23] Dingsøyr T. et al. (2001). Augmenting Experience Reports with Lightweight Postmortem Reviews. En 3rd Intern. Conference on Product Focused Software Process Improvement. Kaiserslautern, Germany.
[24] Aurum A. et al. (2004). Knowledge Management in Software Engineering Education. En IEEE Intern. Conference on Advanced Learning Technologies. Joensuu, Finland.
[25] Li J. (2007). Sharing Knowledge and Creating Knowledge in Organizations: The Modeling, Implementation, Discussion and Recommendations of Web-Log Based Knowledge Management. En International Conference on Service Systems and Service Management. Chengdu, China.
[26] Wongthongtham P. y Chang E. (2007). Ontology Instantiations for Software Engineering Knowledge Management. En IEEE International Symposium on Industrial Electronics. Vigo, Spain.
[27] Georgiades M. et al. (2005). A Requirement Engineering Methodology Based on Natural Language Syntax and Semantics. En 13th IEEE Intern. Conference on Requirements Engineering. Paris, France.
[28] Cuena J. y Molina's M. (2000). The Role of Knowledge Modeling Techniques in Software Development: A General Approach Based on Knowledge Management Tools. International Journal of Human-Computer Studies 52(3), 385-421.
[29] Pilat L. y Kaindl H. (2011). A Knowledge Management Perspective of Requirement Engineering. En 5th Intern. Conference on Research Challenges in Information Science. Gosier, Guadalupe.
[30] White S. (2010). Application of Cognitive Theories and Knowledge Management to Requirements Engineering. En 4th Annual IEEE of Systems Conference. San Diego, USA.
[31] Rouse W. et al. (1992). The Role of Mental Models in Team Performance in Complex Systems. IEEE Transactions on Systems, Man, and Cybernetics 22(6), 1296-1308.
[32] Lambe P. (2007). Organizing Knowledge and Organizational Effectiveness. Chandos.
[33] Uenalan O. et al. (2008). Using Enhanced Wiki-based Solutions for Managing Requirements. En First Inter. Workshop on Managing Requirements Knowledge. Barcelona, Spain.
[34] Supakkul S. et al. (2009). Capturing, organizing, and reusing knowledge of NFRs: An NFR pattern approach. En Second Intern. Workshop on Managing Requirements Knowledge. Washington, USA.

[35] Schmid K. (2009). Reasoning on Requirements Knowledge to Support Creativity. En Second International Workshop on Managing Requirements Knowledge. Washington, USA.
[36] Hevner A. et al. (2004). Design Science in Information Systems Research. Mis Quartely 28(1), 75-105.
[37] Hevner A. (2007). A Three Cycle View of Design Science Research. Scandinavian Journal of Information Systems 19(2), 87-92.
[38] ISO. (1994). ISO 10303-1:1994. Industrial automation systems and integration - Product data representation and exchange - Part 1: Overview and fundamental principles. Intern. ISO.
[39] Heimannsfeld K. y Müller D. (2000). Requirements Engineering Knowledge Management based on STEP AP233. IMW – Institutsmitteilung 25, 73-78.
[40] Scott W. y Cook S. (2003). An Architecture for an Intelligent Requirements Elicitation and Assessment Assistant. En 13th Annual International Symposium INCOSE. Crystal City, USA.
[41] Ratchev S. et al. (2005). Knowledge-enriched Requirement Specification for One-of-a-kind Complex Systems. Concurrent Engineering 13(3), 171-183.
[42] Johnson J. (2006). How does AP233 support a Systems Engineering process (e.g. ANSI/EIA-632)? An update. En 16th Annual International Symposium INCOSE. Swindon, UK.
[43] Hickey A. y Davis A. (2003). Requirements Elicitation and Elicitation Technique Selection: A Model for Two Knowledge-Intensive Software Development Processes. En 36th Hawaii International Conference on Systems Sciences. Hawaii, USA.
[44] Bendjenna H. et al. (2008). Enhancing Elicitation Technique Selection Process in a Cooperative Distributed Environment. Lectures Notes on Computer Science 5025, 23-36.
[45] Vásquez D. et al. (2014). Knowledge management acquisition improvement by using software engineering elicitation techniques. Computers in Human Behavior 30, 721-730.
[46] Tuunanen T. (2003). A New Perspective on Requirements Elicitation Methods. Journal of Information Technology Theory and Application 5 (3), 45-62.
[47] Aranda G. et al. (2005). Choosing groupware tools and elicitation techniques according to stakeholders' features. En International Conference on Enterprise Information Systems. Miami, USA.
[48] Aranda G. et al. (2008). Strategies to minimize problems in global requirements elicitation. En Special issue of best papers presented at workshop doctoral colloquium with one paper selected from CLEI Electronic Journal 11(1), Paper 3.
[49] Huang Y. et al. (2005). Development of an expert system for tackling the public's perception to climate-change impacts on petroleum industry. Expert Systems with Applications 29(4), 817–829.
[50] Mason D. y Pauleen D. (2003). Perceptions of knowledge management: A qualitative analysis. Journal Knowledge Management 7(4), 38-48.
[51] Barkha J. y Sumaira S. (2010). Process support for requirements engineering activities in global software development: A literature based evaluation. En International Conference on Computational Intelligence and Software Engineering. Wuhan, China.
[52] Andrade J. et al. (2006). A Reference Model for Knowledge Management in Software Engineering. Engineering Letters 13(2), 14-21.
[53] Aung Z. y Nyunt K. (2013). Constructive Knowledge Management Model and Information Retrieval Methods for Software Engineering. En K. Buragga y N. Zaman, N. (Eds.), Software Development Techniques for Constructive Information Systems Design (pp. 377-393). IGI Global.
[54] Carreteiro P. et al. (2016). A knowledge management approach for software engineering projects development. En A. Rocha et al. (Eds.), New Advances in Infor. Systems and Tech. (pp. 59-68). Springer.
[55] Rodríguez O. et al. (2004). Applying Agents to Knowledge Management in Software Maintenance Organizations. En Workshop on Agent-Mediated Knowledge Management. Valencia, Spain.
[56] Talib A. et al. (2010). MASK-SM: Multi-Agent System Based Knowledge Management System to Support Knowledge Sharing of Software Maintenance Knowledge Environment. Computer and Information Science 3(2), 52-78.
[57] Anquetil N. et al. (2007). Software maintenance seen as a knowledge management issue. Information and Software Technology 49(5), 515-529.
[58] Pavanasam V. et al. (2010). Knowledge Based Requirement Engineering Framework for Emergency Management System. En Second Intern. Conf. on Comp. Engin. and Applications. Bangalore, India.
[59] Venkatesh K. y Kumar P. (2013). An efficient risk analysis based risk priority in requirement engineering using modified goal risk model. Intern. Journal of Computer Applications 73(14), 15-25.

[60] Olmos K. y Rodas J. (2014). KMoS-RE: Knowledge Management on a Strategy to Requirements Engineering. Requirements Engineering 19, 421-440.
[61] Olmos K. y Rodas J. (2016). A Strategy of Requirements Engineering for Informally Structured Domains. International Journal of Combinatorial Optimization Problems and Informatics 7(2), 49-56.
[62] Chikh A. (2011). A Knowledge Management Framework in Software Requirements Engineering based on the SECI Model. Journal of Software Engineering and Applications 4(12), 718-728.
[63] Mohamed B. et al. (2015). Tacit requirements elicitation framework. Journal of Engineering and Applied Sciences 10(2), 572-578
[64] Hanafiah M. at al. (2015). Towards Developing Collaborative Experience Based Factory Model for Software Development Process in Cloud Computing Environment. Intern. Review on Comp. and Soft. 10(3), 340-350.
[65] Jackson M. (1995). The world and the machine. En 17th international conference on software engineering. Seattle, USA.
[66] Humayoun M. y Qazi A. (2015). Towards Knowledge Management in RE Practices to Support Software Development. Journal of Software Engineering and Applications 8, 407-418.
[67] Valero R. et al. (2013). Decision Support Systems for Incident Management. En 7th Intern. Conf. on Industrial Engineering and Industrial Management. Valladolid, Spain.

CAPÍTULO XIV
Un marco de trabajo para la Gestión del Conocimiento en la Ingeniería de Requisitos[1]

Edgar Serna M.[1]
Alexei Serna A.[1]
Oscar Bachiller S.[2]
[1]Instituto Antioqueño de Investigación
[2]Ingeniero independiente

Gestionar el conocimiento adecuadamente se ha convertido en una necesidad en todo tipo de actividades profesionales, especialmente en aquellas cuyos productos son de alta utilización y divulgación, como en la Ingeniería del Software. Debido a que somos una sociedad software-dependiente este desarrollo tecnológico requiere mejorar los procesos de comprensión y fabricación, por lo que se recomienda utilizar el conocimiento como base para lograrlo. En este capítulo se presenta los resultados de un análisis a las propuestas para gestionar el conocimiento en la Ingeniería de Requisitos, en los que se evidencia que no logran satisfacer las necesidades de los equipos del trabajo en este sentido. Con las prácticas evaluadas se propone un marco integrado para la Gestión del Conocimiento en la Ingeniería de Requisitos, conformado por la conjunción entre las fases del conocimiento y las etapas de la fase, y caracterizado por incluir dos principios del Pensamiento Complejo: Transdisciplinariedad y Multidimensionalidad.

[1] Publicado en inglés en International Journal of Knowledge Management Studies 9(1), 31-50. 2018.

INTRODUCCIÓN

Para la mayoría de investigadores, lo mismo que para la industria, la Ingeniería de Requisitos es la fase más importante y compleja de la Ingeniería del Software. Esto se debe a que en ella se debe comprender totalmente qué es lo que necesita el cliente y definir una estrategia para solucionar su problema con un producto software. Para lograr lo primero el equipo interactúa con todas las partes interesadas para identificar los requisitos, mediante un proceso en el que utiliza diversas técnicas de comunicación y observación. Los requisitos se pueden representar formal o informalmente, siendo más característico lo segundo, debido a que las alternativas de decisión acerca de la funcionalidad y calidad del sistema se verbalizan en lenguaje natural. Todas las actividades involucradas en requieren ingeniería, planificación y gestión que, por cuenta propia, son colaborativas y orientadas a la resolución de problemas. Por lo tanto, la Ingeniería de Requisitos es la fase en la que las partes interesadas descubren, capturan, transforman, capitalizan y utilizan mayor cantidad de datos-información-conocimiento.

De ahí que, en ciertos aspectos, la Ingeniería del Software se asemeja más a la ciencia y las matemáticas que a la ingeniería misma. Porque continuamente se propone y evalúa marcos de trabajo alternativos para mejorar sus fases, lo que se constituye en un proceso científico (experimentación); además, porque la continua definición de capas sobre capas de interpretaciones (abstracción) se desarrolla a través de un proceso similar al progreso de las matemáticas. Estas características la diferencian de otras disciplinas, porque son el resultado directo del rol del software como fuente de complejidad y como medio de elección para implementar nuevas funciones, que se descubren mediante la Gestión del Conocimiento desde la Ingeniería de Requisitos. Esa complejidad y novedad introducen enigmas a los que se enfrenta el equipo de trabajo y, por lo tanto, debe descubrir, crear y gestionar conocimiento.

Para algunos investigadores el problema de no lograr una adecuada Gestión del Conocimiento en la Ingeniería de Requisitos radica en que, con las técnicas actuales, el conocimiento generado en se trata como un sub-producto y, rara vez, se aprovecha [1, 2]. Esto impacta negativamente el desarrollo de software, porque en una industria con restricciones en costos y tiempos que enfatiza en el producto final, la Gestión del Conocimiento tiene poca atención, y se minimiza para alcanzar el objetivo de lograr una entrega a tiempo y dentro del presupuesto.

Un proceso que algunos catalogan como una fórmula simplista en la que, al final, gana la acción de traducir intenciones a código [3, 4]. Además, es auto-destructiva, porque el surgimiento y la inevitable Gestión del Conocimiento no se puede evitar y, al no aprovechar lo que se genera, el equipo estará condenado a redescubrir los mismos errores y a replantear las mismas soluciones una y otra vez. Por otro lado, al no asumir a la Ingeniería de Requisitos como un proceso de alta creación y utilización de conocimiento, se aumenta la probabilidad de tomar decisiones equivocadas en las siguientes fases, porque lo que se cree saber de las necesidades del cliente siempre será incompleto.

En este capítulo se afirma que la Gestión del Conocimiento en la Ingeniería de Requisitos se alimenta y aviva al darle valor a la cadena datos-información-conocimiento, que se genera en la realización de las etapas que conforman esta fase. Porque el producto final no puede ser solamente el sistema en sí, sino también todo el conocimiento generado a través de las diferentes fases del ciclo de vida. Esto incluye documentación completa de la toma de decisiones, las razones, el razonamiento y los criterios que el equipo tuvo en cuenta para lograr una especificación de requisitos completa. Esta especie de *enciclopedia de conocimiento* incluye, entre otras, las lecciones aprendidas, la experiencia adquirida, lo errores cometidos, las soluciones planteadas y

los vínculos al cuerpo de conocimiento más amplio relacionado con el problema. Como solución, se requiere un marco de trabajo para gestionar el conocimiento en esta fase, que le permita al equipo de trabajo construir y alimentar dicha *enciclopedia*.

Esto se puede lograr mediante un marco general para la Gestión del Conocimiento en la Ingeniería de Requisitos, que se pueda utilizar ampliamente y que sirva como base para estructurar un modelo de Gestión de Conocimiento en esta fase. En este trabajo se propone dicho marco de trabajo, con el objetivo de brindarles a los equipos una herramienta de apoyo para gestionar el conocimiento que descubren, analizan e integran en la especificación de requisitos.

1. MARCO REFERENCIAL

1.1 Ingeniería de Requisitos

El término requisito es similar a característica, aunque su alcance es mayor y su enfoque más técnico. Una definición ampliamente aceptada para referirlo se resume en que es una declaración de lo que el sistema debe hacer, cómo debe comportarse, qué propiedades y cualidades debe tener, y las limitaciones que debe satisfacer [5]. Además, se diferencia de *requerimiento* en que éste es sinónimo de necesidad, por lo que se orienta hacia la carencia o falta de algo [6]. Mientras que Aurum y Wohlin [7] consideran que los requisitos son una verbalización de las alternativas de decisión acerca de la funcionalidad y la calidad de un sistema.

Por su parte, la Ingeniería de Requisitos es la fase de la Ingeniería del Software que se ocupa de descubrir, analizar y especificar las propiedades y limitaciones que los clientes necesitan satisfacer con un producto software [8]. Para Nuseibeh y Easterbrook [9] es un proceso para descubrir el propósito del sistema, identificar las partes interesadas y sus necesidades, y documentarlas de forma que se puedan analizar, comunicar e implementar. Por todo esto es que la Ingeniería de Requisitos se considera una etapa compleja en la que, mediante procesos de comunicación exhaustiva con las partes interesadas, se concretan las propiedades y limitaciones reales del sistema. Esta comunicación se desarrolla a través de actividades colaborativas y de resolución de problemas, en las que se intercambia y analiza gran cantidad de conocimiento [10].

1.2 Conocimiento

En términos generales conocimiento es un término que se relaciona con los hechos, la información y las habilidades que adquiere una persona mediante experiencia o formación, que se utiliza popular y cotidianamente en disciplinas tales como las Ciencias Computacionales, donde, a menudo, se mezcla con datos e información. Pero, según Theirauf [11], los datos representan hechos y cifras no-estructurados; la información datos estructurados que tienen utilidad para analizar y resolver problemas críticos; y el conocimiento se obtiene de las personas con base en su experiencia real. Es decir, la información es datos cerca de datos y el conocimiento es información acerca de información. Para Davenport y Prusak [12] conocimiento es una mezcla fluida de experiencias, valores, información e intuición que poseen las personas.

1.3 Gestión del Conocimiento

El término Gestión del Conocimiento se hizo popular a finales de los años 90 y principios del 2000, cuando se convirtió en una palabra de moda. Pero las promesas que se formularon a partir de esta euforia quedaron eclipsadas por una realidad en la que las iniciativas daban lugar a fracasos absolutos de los proyectos [13]. Por todo esto hay que señalar que el mismo hecho de tratar de

definir el término puede ser una amplia fuente de confusión, porque todavía no se ha logrado una definición completamente aceptada [14]. Entre las que se ha publicado se puede referir a Skyrme [15], para quien es la gestión explícita y sistemática del conocimiento vital y los procesos asociados de creación, organización, difusión, uso y explotación en la búsqueda de un objetivo específico. Un enunciado que se acerca a las actividades que se desarrollan en Ingeniería de Requisitos.

2. TRABAJOS RELACIONADOS

- *ISO STEP - AP233* [16]. El objetivo de esta propuesta es proveer un mecanismo capaz de describir la información de un producto a través de su ciclo de vida, independientemente del sistema. Esto la hace adecuada para implementar y compartir bases de datos de productos y archivos, lo mismo que el conocimiento de las personas. Aunque típicamente se utiliza para intercambiar datos e información, también se puede estructurar en la Ingeniería de Requisitos. Quienes la han valorado afirman que proporciona información específica del dominio y que sus protocolos ofrecen conocimiento específico para encontrar y definir requisitos [17]. Scott y Cook [18] sostienen que se trata de un intento riguroso por identificar y hacerle seguimiento al progreso de las etapas de la Ingeniería de Requisitos. Ratchev y sus colegas [19] la adaptan e incorporan en el desarrollo de sistemas, mientras que Johnson [20] afirma que se adapta en el tiempo para clarificar y actualizar requisitos. Como características insalvables todos están de acuerdo en que: 1) no resuelve el problema de la ambigüedad acerca de qué es un requisito; 2) no es suficiente para proporcionar rigor a la gestión del conocimiento, porque los actores no logran una comprensión amplia del problema; y 3) los requisitos se expresan en lenguaje natural al representar el conocimiento descubierto.

- *MTKISDP*. Hickey y Davis [21] presentan una propuesta formal para elicitar requisitos, en la que dan prioridad al conocimiento crítico necesario a partir de: 1) comprender lo que deben realizar los analistas para gestionar el conocimiento, 2) conocer cómo seleccionar una técnica de elicitación, y 3) a medida que se mejora la capacidad para llevar a cabo la elicitación, mejora la probabilidad de que el sistema satisfagan los requisitos. Sobre esta propuesta, Bendjenna et al. [22] afirman que se centra básicamente en la selección de una técnica, y deja la Gestión del Conocimiento en un segundo plano. En este mismo sentido, Vásquez et al. [23] argumentan que los procesos de Gestión del Conocimiento deben abarcar mucho más que identificarlo y capturarlo, por lo que no logra capturar el conocimiento relevante para cada dominio específico. Luego de evaluarla, Tuunanen [24] concluye que, aunque permite representar los usuarios y utiliza grupos y comunidades, no tiene en cuenta la Multidimensionalidad ni la Transdisciplinariedad de los requisitos, por lo que se dificulta gestionar el conocimiento.

- *RMKMSE*. El objetivo de la propuesta de Andrade et al. [25] es lograr que los involucrados en el desarrollo de software accedan al mejor conocimiento posible en el momento oportuno, por lo cual integran la gestión, la clasificación y las características del conocimiento. Para Carreteiro et al. [26] esta propuesta se orienta, esencialmente, a la captura del conocimiento en la fase de requisitos, haciendo uso de la memoria corporativa como fuente principal, pero no aprovecha eficientemente el que poseen las partes interesadas. Rodríguez et al. [27] y Talib et al. [28] dicen que se focaliza principalmente en la participación en el conocimiento y la reutilización de las lecciones aprendidas, dejando de lado otros aspectos de la Ingeniería de Requisitos. Por su parte, Anquetil et al. [29] afirman que no tiene en cuenta el control de la calidad y la corrección, además, que descuida el intercambio en la captura de conocimiento.

- *KBRE*. A través de la integración de sistemas expertos y modelado específico de dominio, Pavanasam et al. [30] proponen una Ingeniería de Requisitos basada en el Conocimiento. Luego

de aplicar esta propuesta, Valero et al. [31] afirman que la toma de decisiones requiere tener conocimiento de múltiples dimensiones, por lo que se debe crear una base de conocimientos, políticas y requerimientos necesarios para tomarlas. Para ellos, KBRE se centra en cómo modelar y utilizar el conocimiento del dominio, pero olvida el conocimiento tácito que poseen las partes interesadas.

- *KMPRE*. Pilat y Kaindl [32] adoptan un proceso en espiral para transformar el conocimiento tácito en explícito y viceversa, con la idea de superar el problema del intercambio de conocimiento y enfatizando en la importancia de elicitar los requisitos a partir de esa reciprocidad. Para Venkatesh y Kumar [33] en esta propuesta predomina la comunicación entre las partes interesadas, lo que permite solucionar parte del problema de la transferencia y el intercambio de conocimiento, cuando esas partes son reacias a hacerlo. Olmos y Rodas [34] sostienen que es incompleta, costosa y que requiere demasiado tiempo en la implementación; además, no logra la transformación ni la transferencia de conocimiento en la dimensión que se requiere en esta fase.

- *KMFSE-SECI* [35]. Utiliza ontologías para describir la especificación y representar el contenido de los requisitos; se sustenta en el modelo SECI. Necesita ir más allá del conocimiento explícito, porque en la Ingeniería de Requisitos el tácito es muy importante [36]. Es algo flexible y adaptable, pero es difícil de aplicar a otras funcionalidades [37].

- *KMoS-RE* [35]. Se orienta a la transformación y transferencia de conocimiento y se caracteriza por: 1) tener en cuenta el modelo de Jackson [38] para el desarrollo de software, 2) basarse en SECI [39], y 3) incorporar métodos para identificar, capturar, indexar y hacer explícito el conocimiento tácito. A este respecto, Humayoun y Qazi [40] sostienen que es una propuesta de gestión de conocimiento sobre una estrategia para Ingeniería de Requisitos, que no tiene en cuenta la complejidad y multidimensionalidad de los requisitos, el dominio informal estructurado, ni las ambigüedades que se deben solucionar.

3. EL CONOCIMIENTO EN LA INGENIERÍA DE REQUISITOS

En la Ingeniería de Requisitos el conocimiento se origina desde dos dimensiones: 1) epistemológica (tácito o explícito), y 2) ontológica (individual o colectivo), en las que existe en forma: 1) cerebral (*embrained*), 2) corporal (*embodied*), 3) codificado (*encoded*), y 4) incrustado (*embedded*). Las interrelaciones entre las dimensiones y las formas del conocimiento en esta fase se aprecian en la Tabla 1.

Tabla 1. Dimensiones y formas del conocimiento en RE

		Dimensión Ontológica	
		Individual	*Colectivo*
Dimensión Epistemológica	*Explícito*	Cerebral	Codificado
	Tácito	Corporal	Incrustado

- El *conocimiento cerebral* es individual y explícito, depende de las destrezas conceptuales y habilidades cognitivas individuales, y puede ser formal, abstracto o teórico.

- El *conocimiento corporal* es individual y tácito, orientado a la acción, práctico y se construye sobre la experiencia práctica. Además, tiene un fuerte componente de automatización y voluntariedad, por lo que su generación y aplicación no encaja fácilmente en un proceso consciente de toma de decisiones. Se desarrolla en un contexto específico y se considera

relevante solamente cuando la utilización soluciona el problema, por lo que la generación no se puede separar de la aplicación.

- El *conocimiento codificado* es colectivo y explícito, y se comparte en forma de símbolos y signos, por lo que se conoce simplemente como *información*. Luego de su codificación se guarda en documentos y otros medios de almacenamiento, que trazan la ruta del comportamiento colectivo. Inevitablemente es simple y selectivo, por lo que no es eficiente para capturar y preservar las habilidades y juicios tácitos individuales.

- El *conocimiento incrustado* es la forma del conocimiento tácito que reside en las normas y reglas compartidas. Se basa en las creencias e interpretaciones colectivas que hacen posible una comunicación efectiva entre las comunidades de práctica. Además, es de relación específica, contextual y disperso, orgánico y dinámico.

Debido a esas diferentes formas el conocimiento en la Ingeniería de Requisitos se puede manifestar como explícito o como tácito (*dimensión epistemológica*) y sus diferencias radican en:

1. *La codificación y la transferencia*. El conocimiento explícito se puede codificar, abstraer y almacenar en el mundo objetivo (contexto del problema), además de comprender y compartir sin la asistencia de un *conocedor*, porque es fácil de comunicar y transferir. Mientras que el tácito es personal, intuitivo, desarticulado y reside en un mundo no-objetivo (creencias, interpretaciones, mitos, valoraciones del problema), por lo que no es fácil comunicarlo, comprenderlo ni utilizarlo sin la asistencia de un *conocedor*. Desde esta dimensión, y mediante técnicas de comunicación e interpretación comunes para todas las partes interesadas, el equipo transfiere, comparte y analiza el conocimiento explícito en el espacio-tiempo de esta fase. Mientras que el tácito requiere técnicas y métodos específicos para lograr que cada conocedor comparta lo que sabe.

2. *La consecución y la compilación*. El conocimiento explícito se consigue mediante deducciones lógicas, que utiliza el equipo de trabajo a través de la observación y el análisis, y compilado desde estudios formales que realiza con las partes interesadas sobre documentos y normas. Mientras que el conocimiento tácito se adquiere a través de prácticas experimentales con los actores individuales, y se compila mediante observación y deducción.

3. *La acumulación y la apropiación*. El explícito se puede agregar desde una única ubicación, almacenar en forma de objetivos y apropiar sin la intervención del actor conocedor. Pero debido a que es personal y contextual, el tácito se está distribuido y no se puede acumular ni apropiar con facilidad, por lo que se debe utilizar técnicas diferentes con cada conocedor.

Asimismo, el conocimiento en la Ingeniería de Requisitos también se puede manifestar de forma individual o colectiva (*dimensión ontológica*). En la primera, el conocimiento reside en la mente y en las habilidades y destrezas de las personas, por lo que se considera de su propiedad y es autónomo. Se caracteriza por ser inevitablemente especializado y de dominio específico, puede ser transferible, pero se desplaza con el propietario, por lo que genera retención y acumulación. Por su parte, el conocimiento colectivo es el que se comparte y distribuye entre diferentes individuos y se almacena en las normas, procesos, rutinas, directrices y reglas que utilizan para solucionar problemas.

Desde finales de siglo los investigadores de la Ingeniería del Software pusieron su atención en el estudio del conocimiento involucrado en el desarrollo de software, y publicaron enfoques y

herramientas con el objetivo de mejorar su gestión en el ciclo de vida [41, 42]. Algunos se orientaron a la fábrica de experiencias [43], el diseño de patrones [44], a los procesos en equipo [45], el conocimiento enriquecido [46], Gestión del Conocimiento basado en procesos [47], los procesos personales [48] y la gestión racional [49], pero, a excepción de esta última, los demás le prestan poca atención a la Gestión del Conocimiento en la Ingeniería de Requisitos, orientándose principalmente a las fases de diseño, implementación y mantenimiento.

Por otro lado, los estudios empíricos en este mismo tema poco se han enfocado en la Gestión del Conocimiento de las partes interesadas. Por ejemplo, Ko et al. [50] estudian las necesidades de información de los desarrolladores; Sillito et al. [51] analizan la necesidad de conocimiento en los desarrolladores para el mantenimiento de software, y Robillard [52] estudia los obstáculos que enfrentan los desarrolladores cuando reutilizan el conocimiento de los componentes. Ninguno se enfoca en el conocimiento de las partes interesadas cuando se elicitan o implementan los requisitos, una cuestión importante para comprender sus necesidades y proporcionarles apoyo eficaz para la comprensión del problema.

Además, en los procesos de la Ingeniería del Software y en la comunidad de la Ingeniería de Requisitos se han popularizado las herramientas de Gestión del Conocimiento, tales como los sistemas de gestión documental [53], los sistemas de recomendación [54, 55], las herramientas de búsqueda y recuperación de información [56], los *wikis* [57] y los repositorios basados en ontologías [58]. Todo esto genera confusión en los equipos de trabajo, porque se les ofrece muchas opciones, pero ninguna experiencia práctica que les permita seleccionar la más adecuada para el problema que intentan resolver. Por eso es que se requiere un marco de trabajo integrado para gestionar el conocimiento en la Ingeniería de Requisitos, que se pueda utilizar ampliamente y que sirva como base para estructurar un modelo de Gestión del Conocimiento en esta fase.

4. MARCO DE TRABAJO PARA LA GESTIÓN DEL CONOCIMIENTO EN LA INGENIERÍA DE REQUISITOS

La Ingeniería de Requisitos es una fase de la Ingeniería del Software en la que la interacción con las partes interesadas es permanente, porque es aquí donde el quipo reconoce lo que ellas saben del problema y lo que necesitan del sistema. En las diferentes etapas que la conforman (Figura 1) se realiza intercambios permanentes de datos-información-conocimiento, que el equipo debe gestionar adecuadamente para lograr la estructuración de un buen documento de especificación de requisitos.

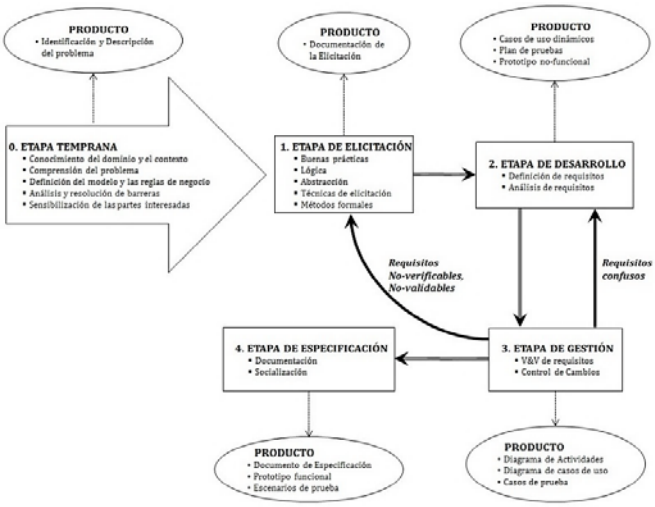

Figura 1. MoDeMaRE [59]

En la Etapa Temprana se realiza conversatorios con las partes interesadas para conocer el dominio y el contexto del problema, y para abonar el terreno del trabajo con ellas; en la Elicitación se aplica técnicas para dialogar con todos los actores y descubrir lo que cada uno sabe, y puede aportar para encontrar los requisitos desde sus dimensiones; en la Etapa de Definición, y mediante interacciones entre todas las disciplinas involucradas, se define y analiza los requisitos elicitados; en la Gestión se verifica y valida los requisitos desarrollados para gestionar los cambios resultantes y, en la Etapa de Especificación, se estructura el documento de la especificación con la información y descripción de los requisitos.

En el proceso, además de los requisitos que las partes interesadas comparten, también hay que localizar los que no pueden, o no quieren, compartir. Estos requisitos se caracterizan por ser difíciles de expresar, convertir, comunicar y compartir; solamente se relacionan con el dominio del sistema; son experiencias o conocimientos propios de las personas; son difíciles de codificar y articular y, generalmente, se expresan de forma vaga y cruda desde la visión del poseedor. Ambos tipos de requisitos se clasifican, respectivamente, como *explícitos* y *tácitos*, y de forma general los segundos pueden ser más espinosos de obtener de las personas de manera directa y verbal, o de descubrir su existencia mediante la observación.

Por eso se requiere un marco de trabajo, porque además de comprender el problema y estructurar una posible solución, el equipo debe tratar con personas que quieren colaborar y con otras que apenas si lo hacen. El conocimiento, explícito o tácito, primero se debe descubrir a partir de su portador, para gestionarlo mediante análisis, discusión y comprensión colaborativos. Un entorno de este tipo debe ser iterativo, en el que las partes interesadas reconocen el lenguaje técnico, y el equipo se familiariza con las necesidades y formas de comunicación que utilizan. Pero esto no es tan simple como parece, porque el proceso implica trabajar con personas de diferentes disciplinas, con puntos de vista disímiles sobre el problema y el sistema, con aspiraciones complicadas, con interpretaciones arraigadas, y muchas más características. Esta situación genera problemas de comunicación que impactan la calidad de la especificación de los requisitos.

En la práctica la Ingeniería de Requisitos se desarrolla en un ambiente de conocimiento intensivo, por lo que el equipo de trabajo debe gestionarlo para aprovecharlo eficientemente en pro de construir el sistema. En esta fase del ciclo de vida el conocimiento es diverso y creciente, por lo que los equipos tienen problemas para identificarlo, ubicarlo y encontrar la mejor forma de gestionarlo. Tomando como base a MoDeMaRE [59], en la Figura 2 se propone el marco de trabajo integrado de la proximidad entre las fases del conocimiento y las etapas de la Ingeniería de Requisitos, cuyo objetivo es encontrar el conocimiento explícito y descubrir y transformar el táctico, para generar un documento de especificación de requisitos sólido y fiable.

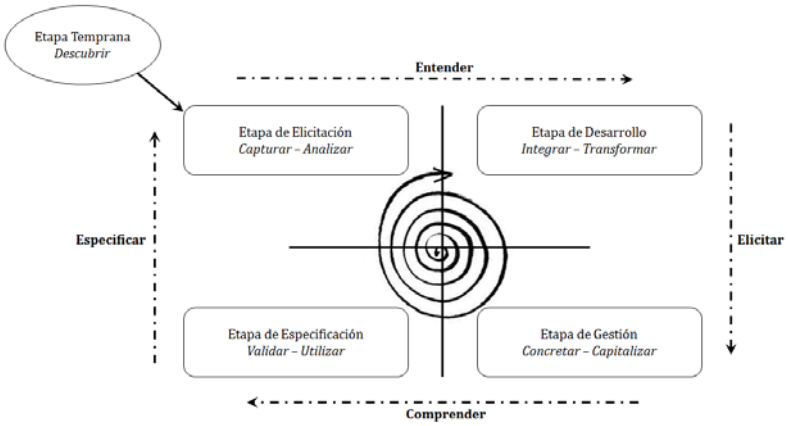

Figura 2. Marco de trabajo propuesto

4.1 Descubrir

El objetivo de esta fase es seleccionar las herramientas/prácticas apropiadas al contexto y dominio del problema y del sistema, para descubrir la cadena datos-información-conocimiento que poseen las personas, la organización y/o los sistemas involucrados. Una característica inmersa es detectar los patrones que sigue esta cadena, porque, generalmente, están ocultos, y es necesario aplicar prácticas específicas para cada tipo esperado. El contenido de los documentos, los registros, la multimedia y los repositorios de datos representan el conocimiento explícito y son dimensiones origen para estos requisitos. Teniendo en cuenta las diferentes dimensiones (socio-cultural, político-institucional, administrativa, tecnológica y disciplinar), desde las que se originan los requisitos, hay que reconocer los patrones y las relaciones de los datos, el texto o las imágenes. En estos casos la observación es la herramienta más eficiente, pero la minería de datos y de texto también son útiles en esta tarea.

Descubrir el conocimiento tácito es más complejo y requiere una visión de gestión transdisciplinar. El hecho de que esté oculto no quiere decir que sea imposible de encontrar, pero exige un poco más de habilidades y destrezas en el equipo de trabajo. Nuevamente, la observación es una poderosa herramienta, pero existe otras alternativas cualitativas y cuantitativas, tales como encuestas, cuestionarios, entrevistas individuales y grupales, grupos focales y análisis de redes. Por otro lado, las personas dejan una huella de su conocimiento y experiencia en cada opinión o participación, por lo que se recomienda seguirla para tratar de descifrarla, sobre todo cuando esas personas se muestran reacias a colaborar directamente. Esto es fundamental con aquellos individuos que se consideran *fuentes primarias* de los requisitos, es decir, son los *únicos* que conocen los procesos centrales del problema y el sistema.

Tanto el conocimiento explícito como el tácito pueden existir en dimensiones internas o externas al problema, la organización y/o el sistema. Esto se evidencia en las asociaciones o redes en las que participan, formal o informalmente, las partes interesadas, tales como proveedores, clientes o usuarios, y que se relacionan de alguna manera con el problema. Aquí hay que ser cautos, porque muchas de esas redes se basan en la confianza entre los actores e intentar romperla puede generar mayor hermetismo. La recomendación es adoptar prácticas orientadas a descubrir lo que cada individuo sabe, sin violentar sus relaciones o decisiones. Por ejemplo, solicitar un documento formal en lenguaje técnico, en el que describa los aspectos más importantes de su trabajo, podría ofrecer la minería suficiente para descubrir la cadena relacionada con los requisitos embebidos, además de poder seguir su huella.

4.2 Capturar – Analizar

El objetivo de esta fase es clasificar y comprender la cadena datos-información-conocimiento descubierta, de tal manera que se pueda aprovechar en las demás fases. La idea es establecer buenas prácticas de análisis para decantar el conocimiento que realmente será útil para elicitar los requisitos. Es muy importante que el equipo trabaje como tal, porque debe discutir lo descubierto y llegar a acuerdos de comprensión. Para lograrlo, hay que tener cuenta los diversos y diferentes puntos de vista e interpretaciones de las partes interesadas, lo mismo que las dimensiones y las disciplinas involucradas, es decir, debe obrar como un *gestor de conocimiento* para alcanzar la comprensión necesaria.

Existe diferentes maneras para lograrlo, pero, debido a que el proceso involucra capturar, analizar y comprender el conocimiento descubierto, se recomienda utilizar ontologías [60, 61] y taxonomías [62]. Además, al funcionar como un gestor de conocimiento el equipo puede aplicar

flujos y medidas de rendimiento, con lo que obtendría un mapeo de la cadena descubierta. Sin perder de vista que de lo que se trata es de capturar el conocimiento que indique la presencia de requisitos, el análisis es una tarea que, si no se organiza como proceso, puede complicar dicha captura. El equipo debe tener la habilidad suficiente para filtrar lo que no aporta al objetivo de la fase, es decir, en este momento ha descubierto gran cantidad de datos-información-conocimiento, tanto explícito como tácito, pero no todo es útil ni comprensible. En esta fase, solamente se mapea el conocimiento que el equipo determina como esencial.

Por otro lado, es común que se confunda términos como entender, comprender, saber y conocer, pero existe una amplia diferencia en sus significados. Sin entrar en la controversia que se pueda generar, en este trabajo solamente interesa aclarar qué se entiende por comprender: las personas saben y entienden datos y conocen información, pero los comprenden únicamente cuando los convierten en conocimiento, es decir, cuando les dan significado y, entonces, generan aprendizaje. Por eso es que en esta fase se construye conocimiento a partir de los datos y la información, pero analizados mediante un proceso de selección, organización e integración, a lo que el equipo debe sumar lo que individualmente sabe, entiende y comprende del problema y el sistema. El proceso que se propone en esta fase es el siguiente:

1. *Realizar una depuración de los resultados del análisis* para diferenciar los mitos, las creencias, las curiosidades y las interpretaciones de las verdaderas necesidades a satisfacer. Hay que recordar que cada disciplina presenta una visión individual del contexto del problema y el sistema, por lo tanto, se integran para llegar a una comprensión global de la situación.

2. *Determinar cuánto de la cadena datos-información-conocimiento se comprende realmente*. El equipo toma lo que entiende, sabe y conoce desde las disciplinas y las dimensiones, para crear un mapa de la comprensión del problema y de las necesidades del sistema.

3. *Relacionar la comprensión del contexto con la situación actual*. Cada cadena analizada se ubica en un momento anterior del contexto, por lo que hay que relacionarla con el momento actual y proyectarla a un momento posterior, porque de otra manera se estará intentando solucionar el problema con base en la comprensión de un contexto anterior.

4. *Darle significado al conocimiento contextualizado*. Mediante trabajo en equipo, e integrando todas las partes interesadas, se modela la comprensión, de tal forma que todos concuerden sobre el aprendizaje logrado.

5. *Convertir el aprendizaje*, desde las disciplinas y las dimensiones, en una lista de requisitos básicos que el sistema debe satisfacer. Esta comprensión es fundamental para crear el primer prototipo de la solución.

4.3 Integrar – Transformar

Hasta el momento la Gestión del Conocimiento se ha llevado a cabo conjuntamente entre los actores, pero todavía persiste interpretaciones individuales/disciplinares que se deben sintetizar en una representación única. Para esto hay que involucrar al cuerpo del conocimiento externo a la situación-problema, es decir, tener en cuenta otras dimensiones que tienen representatividad en el problema y en el sistema. Este es un proceso de integración en el que el conocimiento existente interactúa con el externo, para lograr un modelo de conocimiento amplio y acordado desde las disciplinas y las dimensiones involucradas. De esta manera se logra un aprendizaje adquirido e integrado acerca del problema, además del contexto y dominio del sistema.

Es importante no olvidar que integración y transformación transdisciplinar significa integrar personas, conocimientos y tecnologías que pertenecen a los diferentes dominios y contextos en los que el sistema existe o con los que tiene relación. Las dos actividades para lograrlo son: 1) encontrar un lenguaje común, necesario para dialogar sobre necesidades y requisitos, sobre opciones y caminos, y sobre el contenido del documento de la especificación, y 2) diseñar una metodología para la integración. No se debe confundir la integración transdisciplinar con los intentos de establecer un conocimiento de la situación como un todo, porque esto sería como volverlo a parcelar en cada disciplina y dimensión origen.

Hay que lograr un lenguaje común, porque de esta manera se rompe las barreras de entendimiento entre las disciplinas y porque las personas pueden expresar sus interpretaciones, para que todos las puedan comprender y/o transformar. Una forma de hacerlo es encontrar los conceptos comunes a las disciplinas de amplio conocimiento para cualquier profesional, por ejemplo, desde las matemáticas y la teoría de sistemas, o conceptos básicos de gestión y administración. En este sentido es conveniente consultar sobre las diferentes escuelas de sistemas de pensamiento.

Luego de alcanzar el lenguaje común, la siguiente actividad es integrar métodos y teorías en una metodología adecuada para transformar el conocimiento. Este es un trabajo de equipo y, aunque el resultado podría no ser una teoría unificada, se crea un mosaico en el que los métodos se conectan a través de los conceptos compartidos que el equipo determinó desde el lenguaje común. El producto final de esta fase es un listado de requisitos integrados y transformados en un lenguaje técnico, que todos los participantes comprenden y enriquecen desde sus disciplinas con el conocimiento y experiencia que poseen.

4.4 Concretar – Capitalizar

En relación con el conocimiento existen dos conceptos que se han popularizado y estudiado desde hace tiempo: lo *abstracto* y lo *concreto*. El primero se refiere a una especie de interpretación aislada desde una realidad, que para representarse requiere imaginación e ingenio. Esto sucede continuamente en la Ingeniería de Requisitos, porque las partes interesadas construyen, desde su disciplina y experiencia, interpretaciones propias del problema y del sistema, con el inconveniente de que llegan a aceptarla como conocimiento. Esto que crea barreras de comunicación que se deben derrumbar para alcanzar una verdadera comprensión de la situación. En esta fase de la gestión del conocimiento el quipo todavía mantiene interpretaciones abstractas de los requisitos del sistema, por lo que hay que concretar el conocimiento a partir del lenguaje común y la metodología estructurada en la fase anterior.

Por otro lado, el conocimiento concreto se refiere a la imagen que el equipo moldea del problema y del sistema, es decir, una amalgama de aspectos y propiedades singulares que comprenden, porque los consideran como momentos de un todo, determinables por el contenido específico del mismo problema. Aquí es importante tener presente que las personas no descifran o resuelven problemas del mundo real, sino representaciones abstractas de ellos, y en la Ingeniería de Requisitos es común que involucren un proceso de reducción en el que sacrifican aspectos, que consideran poco relevantes, para concentrarse en variables que creen le dan un alcance más general a su interpretación. Esto sucede porque dicha imagen la construyen desde su disciplina, y desde ella encuentran las respuestas que pueden y necesitan, con el agravante de que las aceptan con carácter universal. Por eso es importante que el equipo trabaje desde lo multidimensional y transdisciplinar, porque de esta forma puede concretar el conocimiento integrado y transformado.

Además, en la Ingeniería de Requisitos se debe tener presente que los problemas se analizan en situaciones, sin utilizar definiciones generales, porque cada parte del mismo se vincula y pertenece a una situación específica, y solamente mediante una compresión concreta de esa vinculación se puede llegar a un análisis acertado del conocimiento involucrado. Al comprender el conocimiento y encontrar el lenguaje común para expresarlo, el equipo podrá concretarlo, es decir, capitalizar el disciplinar para encontrar una solución al problema. Esto se debe a que la suma de experiencias, interpretaciones y habilidades de todos los actores posibilita el desarrollo potencial de ideas, para sintetizar lo que realmente se conoce sobre el problema y de las necesidades que el sistema debe satisfacer.

4.5 Validar – Utilizar

En términos generales la validación es un proceso para asegurar que algo es correcto o se ajusta a cierto nivel de exigencia. En el marco presentado en este trabajo el término se utiliza para garantizar que la cadena datos-información-conocimiento se ubica dentro de los límites aceptados de la comprensión del problema y/o el sistema. A este proceso se le debe considerar crítico en la Ingeniería de Requisitos, porque el conocimiento y las interpretaciones son componentes cruciales que se deben validar. La mayoría de autores concuerda en que este proceso se desarrolla mediante dos actividades: 1) de *verificación*: que tenga la intención de alcanzar una estructura correcta para el conocimiento, y 2) de *evaluación*: que tenga la intención de demostrar la utilidad de ese conocimiento para llegar a conclusiones correctas, es decir, encontrar los requisitos a satisfacer.

Tradicionalmente, las actividades de verificación se apoyan en técnicas específicas para facilitar la realización del procedimiento, tales como tablas de decisión o árboles, algunas formas gráficas, e incluso técnicas de aprendizaje automático. Debido a que gran parte del conocimiento concreto está representado en las reglas de negocio de la empresa, se recomienda formalizar estos requisitos para facilitar interpretaciones unificadas. Por otro lado, las actividades de evaluación se sustentan en pruebas que permitan llegar a la conclusión de que el conocimiento es correcto. Tanto la verificación como la evaluación son labores transdisciplinares, en la que se tiene en cuenta las dimensiones de la cadena datos-información-conocimiento. En esta fase la cadena se debe romper para trabajar únicamente con el conocimiento concreto y capitalizado, porque el objetivo es redactar el documento de la especificación de requisitos, es decir, hay que utilizar conocimiento validado.

Lo primero que hay que establecer son los objetos concretos que representa ese conocimiento, es decir, materializar la comprensión del problema. Esto se logra a partir del conocimiento del dominio y el contexto del mismo, así como del modelo que representa el conocimiento validado. Posteriormente, se define el alcance de la utilización, o sea, demarcar los límites transdisciplinares referenciales. Esto se debe a que el conocimiento disciplinar se ve limitado en las interpretaciones individuales, entonces se requiere las relaciones entre sus fronteras para delimitar la utilización al conjunto completo de disciplinas involucradas en el problema y el sistema. En todo caso, los criterios de utilización del conocimiento deben: 1) ser conformes a los procedimientos de las demás fases, 2) tener dependencia específica, 3) ser independientes de los criterios de evaluación, 4) tener unidad práctica, y 5) ser complementarios entre sí.

5. VALIDACIÓN Y VERIFICACIÓN DEL MARCO DE TRABAJO

Verificación y Validación V&V es una metodología que permite encontrar las certezas y los errores de propuestas, modelos y marcos propuestos en cualquier disciplina. Cuantificar la confianza y

exactitud de lo propuesto proporciona información importante para tomar decisiones sobre su aplicabilidad y funcionalidad. Pero, cualquiera que sea el paradigma de modelado o la técnica de solución que se aplique, las medidas de rendimiento solamente tendrán valor si el marco se aplica en una buena representación del sistema real. Por supuesto, lo que constituye una buena propuesta es subjetivo, pero desde un punto de vista lógico los criterios para juzgar sus bondades se basan en que los resultados de la Verificación y la Validación corresponden a los esperados en su aplicación real.

En este sentido, la exactitud se observa como la ausencia sistemática y aleatoria de errores, lo que se denomina fidelidad y precisión. En términos generales la Verificación debe demostrar que el marco es verdadero, mientras que la Validación que es digno de crédito. Para ello existen principios que se deben tener en cuenta: 1) todos los marcos son erróneos, aunque algunos son más útiles que otros, 2) no se debe considerar como el único en su género, y 3) es necesario comprobar cuidadosamente que se ajusta a lo esperado. Lo más importante en este sentido es que un marco de trabajo como el que se propone en esta investigación no se puede validar y verificar como un todo, porque su contexto de aplicación es complejo y el proceso requiere un tiempo prolongado, características que no son fáciles de conseguir para experimentarlo directamente en la industria. Por eso se recomienda simularlo individualmente para cada uno de sus componentes, de forma incremental y aplicando un seguimiento *top-down*, como se propone en la Figura 3.

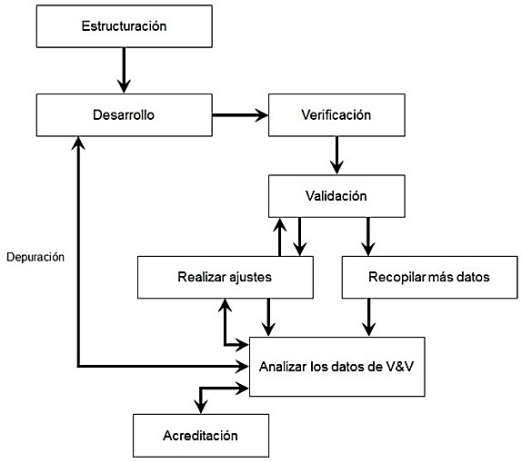

Figura 3. Proceso de V&V para el marco de trabajo propuesto (Adaptado de Law [63])

Este proceso se simuló completamente sobre la Ingeniería de Requisitos en un sistema de administración de seguridad de ingreso en una entidad bancaria; con el análisis a los resultados se realizaron los ajustes y depuraciones necesarias para cada una de las etapas de esta fase (Figura 1) y las fases del conocimiento (Figura 2). El centro de estas actividades es la recopilación y análisis de la cadena datos-información-conocimiento hallada en la aplicación del marco a cada etapa, porque influyen directamente en la toma de decisiones para: 1) ajustar los parámetros asociados para alcanzar mayor comprensión del conocimiento adquirido, 2) rediseñar el marco con base en los resultados de la simulación, y 3) acreditar la aceptabilidad del marco con respecto a cada fase-etapa simulada. El objetivo de la simulación del marco fue encontrar:

1. *Validez aparente*. Se solicitó a varios especialistas en la temática, con conocimientos sobre Gestión del Conocimiento e Ingeniería de Requisitos, que hicieran una evaluación conceptual para determinar si el marco y/o su comportamiento eran lógicos y razonables. Se recibieron los aportes y sugerencias, y la conclusión es que es lógico y viable, pero que en la industria el proceso de aplicación y contrastación requiere tiempo.

2. *Validación predictiva*. Mediante aplicación simulada se utilizó el marco para predecir (pronosticar) su comportamiento y valores de respuesta en un contexto real. Con los resultados se realizaron los ajustes necesarios y cada etapa-fase se puso a punto, antes de iniciar la siguiente. En la fase 0, Descubrir Conocimiento, se encontró que determinar el conocimiento explícito es complicado, porque en esta etapa las partes interesadas tienen predisposiciones hacia el nuevo sistema, y esas barreras se deben identificar y derrumbar antes de iniciar la fase de descubrimiento. Esto se corrigió en el marco, de tal manera que primero se realizó la sensibilización para luego iniciar el descubrimiento del conocimiento explícito. En esta fase todavía no se pretende encontrar masivamente conocimiento tácito en las personas, lo que se corrobora en los resultados. Luego de realizar la simulación de forma progresiva e iterativa en cada una de las etapas la Ingeniería de Requisitos y de las fases de la Gestión del Conocimiento, se hicieron los ajustes y las depuraciones sugeridas, demostrando al final que el marco de trabajo supera la validación predictiva.

3. *Divulgación y Acreditación*. Se refiere a la publicación del marco y a la búsqueda de una aplicación industrial. Este objetivo se logra con la aceptación del artículo en la revista y su posterior edición. Además, hasta el momento los colegas en las universidades empiezan a experimentarlo en sus asignaturas y a capacitar a los estudiantes para que lo pongan en producción. De la misma manera, los autores iniciaron contactos con la industria para brindar asesoría en la aplicación del marco e iniciar su validación en problemas y sistemas reales. Se ha logrado una adecuada comprensión progresiva de la propuesta, debido a que continuamente se socializa y se repite la simulación-validación predictiva.

6. CONCLUSIONES

La Ingeniería de Requisitos se considera la fase más importante y compleja del desarrollo de software, porque en ella se descubre y especifica las necesidades que debe satisfacer el sistema. Es un proceso en el que las partes interesadas se comunican y comparten sus ideas y apreciaciones acerca de lo que necesitan y esperan de la solución que el equipo les presentará. Por eso es la fase en la que se presenta mayor comunicación entre ellos y en la que se comparte continuamente la cadena datos-información-conocimiento.

Aunque esta característica se reconoce en la comunidad y la industria, la Gestión del Conocimiento que se realiza todavía no es satisfactoria para concretar las necesidades de las partes interesadas y del sistema mismo. La revisión a los trabajos relacionados demuestra que las propuestas se quedan cortas en este aspecto y pocas logran acercarse a una verdadera gestión de la cadena. Algunas se especializan en la elicitación de requisitos, otras en el desarrollo de los mismos y algunas en la especificación, pero no son homogéneas a lo largo de todas las etapas de esta fase del ciclo de vida.

Debido a esto, y luego de analizar lo que se ha publicado y experimentado hasta el momento en relación con la Gestión del Conocimiento en la Ingeniería de Requisitos, en este trabajo se propone un marco de trabajo integral para lograr ese objetivo. Se toma algunas prácticas de las propuestas de la revisión y se enriquecen con principios del Pensamiento Complejo, aspectos que no han tenido en cuenta otros investigadores.

REFERENCIAS

[1] Wan J. et al. (2010). Research on Knowledge Creation in Software Requirement Development. Software Engineering & Applications 3(5), 487-494.

[2] Silva R. y Dasilva A. (2015). Engineering Requirements for crowds. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 5(2), 17-20.
[3] Alvear A. y Quintero G. (2015). Integrating software development techniques, usability, and agile methodologies. Revista Actas de Ingeniería 1, 94-103.
[4] Sutcliffe A. y Sawyer P. (2013). Requirements Elicitation: Towards the Unknown Unknowns. En 21st IEEE International Requirements Engineering Conference. Rio de Janeiro, Brasil.
[5] Clements P. y Northrop L. (2006). A framework for software product line practice. Carnegie Mellon.
[6] Pagani J. (2013). ¿Requisitos o Requerimientos? Investigación IT.
[7] Aurum A. y Wohlin C. (2003). The fundamental nature of requirements engineering activities as a decision-making process. Information and Software Technology 45(14), 945–954.
[8] Davis A. (2003). The art of requirements triage. Computer 36(3), 42–49.
[9] Nuseibeh B. y Easterbrook S. (2000). Requirements engineering: A roadmap. En Conference on the future of software engineering. Limerick, Ireland.
[10] Maalej W. y Thurimella A. (2013). An Introduction to Requirements Knowledge. En W. Maalej y A. Thurimella (Eds.), Managing Requirements Knowledge (1-20). Springer.
[11] Thierauf R. (1999). Knowledge management systems for business. Praeger.
[12] Davenport T. y Prusak L. (2000) Working Knowledge: How organizations manage what they know. Harvard Business School Press.
[13] Akhavan P. et al. (2005). Exploring failure-factors of implementing knowledge management systems in organizations. Journal of Knowledge Management Practice 6, 1-8.
[14] Frost A. (2014). Knowledge management. Recuperado: http://www.knowledge-management-tools.net/
[15] Skyrme D. (2011). Knowledge Management: Definition. Recuperado: http://www.skyrme.com/kmbasics/definition.htm
[16] ISO. (1994). ISO 10303-1:1994. Industrial automation systems and integration -- Product data representation and exchange -- Part 1: Overview and fundamental principles. International Organization for Standardization.
[17] Heimannsfeld K. y Müller D. (2000). Requirements Engineering Knowledge Management based on STEP AP233. IMW - Institutsmitteilung 25, 73-78.
[18] Scott W. y Cook S. (2003). An Architecture for an Intelligent Requirements Elicitation and Assessment Assistant. En 13th Annual International Symposium INCOSE. Crystal City, USA.
[19] Ratchev S. et al. (2005). Knowledge-enriched Requirement Specification for One-of-a-kind Complex Systems. Concurrent Engineering 13(3), 171-183.
[20] Johnson J. (2006). How does AP233 support a Systems Engineering process (e.g. ANSI/EIA-632)? An update. En 16th Annual International Symposium INCOSE. Swindon, UK.
[21] Hickey A. y Davis A. (2003). Requirements Elicitation and Elicitation Technique Selection: A Model for Two Knowledge-Intensive Software Development Processes. En 36th Hawaii International Conference on Systems Sciences. Hawaii, USA.
[22] Bendjenna H. et al. (2008). Enhancing Elicitation Technique Selection Process in a Cooperative Distributed Environment. Lectures Notes on Computer Science 5025, 23-36.
[23] Vásquez D. et al. (2014). Knowledge management acquisition improvement by using software engineering elicitation techniques. Computers in Human Behavior 30, 721-730.
[24] Tuunanen T. (2003). A New Perspective on Requirements Elicitation Methods. Journal of Information Technology Theory and Application 5 (3), 45-62.
[25] Andrade J. et al. (2006). A Reference Model for Knowledge Management in Software Engineering. Engineering Letters 13(2), 14-21.
[26] Carreteiro P. et al. (2016). A knowledge management approach for software engineering projects development. In A. Rocha et al. (Eds.), New Advances in Inform. Syst. and Tech. (pp. 59-68). Springer.
[27] Rodríguez O. et al. (2004). Applying Agents to Knowledge Management in Software Maintenance Organizations. Semantic Scholar.
[28] Talib A. et al. (2010). MASK-SM: Multi-Agent System Based Knowledge Management System to Support Knowledge Sharing of Software Maintenance Knowledge Environment. Comp. and Infor. Science.
[29] Anquetil N. et al. (2007). Software maintenance seen as a knowledge management issue. Information and Software Technology.

[30] Pavanasam V. et al. (2010). Knowledge Based Requirement Engineering Framework for Emergency Management System. En Second International Conference on Computer Engineering and Applications. Bangalore, India.
[31] Valero R. et al. (2013). Decision Support Systems for Incident Management. En 7th Intern. Conf. on Industrial Engineering and Industrial Management. Valladolid, Spain.
[32] Pilat L. y Kaindl H. (2011). A Knowledge Management Perspective of Requirements Engineering. En Fifth International Conference on Research Challenges in Information Science. Gosier, Guadeloupe.
[33] Venkatesh K. y Kumar P. (2013). An Efficient Risk Analysis based Risk Priority in Requirement Engineering using Modified Goal Risk Model. Inter. Journal of Computer Applications 73(14), 15-25.
[34] Olmos K. y Rodas J. (2014). KMoS-RE: Knowledge management on a strategy to requirements engineering. Requirements Engineering 19(4), 421–440.
[35] Chikh A. (2011). A Knowledge Management Framework in Software Requirements Engineering based on the SECI Model. Journal of Software Engineering and Applications 4(12), 718-728.
[36] Mohamed B. et al. (2015). Tacit requirements elicitation framework. ARPN Journal of Engineering and Applied Sciences 10(2), 572-578.
[37] Hanafiah M. at al. (2015). Towards Developing Collaborative Experience Based Factory Model for Software Development Process in Cloud Computing Environment. Intern. Review on Comp. and Soft. 10(3), 340-350.
[38] Jackson M. (1995). The world and the machine. En 17th international conference on software engineering. Seattle, USA.
[39] Nonaka I. y Takeuchi H. (1995). The knowledge-creating company: How Japanese companies foster creativity and innovation for competitive advantage. Harvard Business Review 69(6), 96-104.
[40] Humayoun M. y Qazi A. (2015). Towards Knowledge Management in RE Practices to Support Software Development. Journal of Software Engineering and Applications 8, 407-418.
[41] Bjørnson F. y Dingsøyr T. (2008). Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. Information and Software Technology 50(11),1055–1068.
[42] Babar M. (2009). Supporting the Software Architecture Process with Knowledge Management. En P. Lago et al. (Eds), Software architecture knowledge management: Theory and practice. Springer.
[43] Basili V. et al. (1994). The experience factory - Encyclopedia of software engineering. Willey.
[44] Gamma E. et al. (1995). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.
[45] Humphrey W. (1999). Introduction to the team software process. Addison-Wesley.
[46] Basili V. et al. (2001). An experience management system for a software engineering research organization. En 26th annual NASA Goddard Software engineering workshop. Greenbelt, Maryland.
[47] Holz H. (2003). Process-based knowledge management support for software engineering. Doctoral dissertation. University of Kaiserslautern.
[48] Humphrey W. (2005). PSP: A self-improvement process for software engineers. Addison-Wesley.
[49] Dutoit A. et al. (2006). Rationale management in software engineering. Springer.
[50] Ko A. et al. (2007). Information needs in collocated software development teams. En 29th international conference on software engineering. Minneapolis, USA.
[51] Sillito J. et al. (2008). Asking and answering questions during a programming change task. Transactions on Software Engineering 34(4), 434–451.
[52] Robillard M. (2009). What makes APIs hard to learn? Answers from developers. IEEE Software 26(6), 27–34.
[53] Rus I. et al. (2001). Knowledge management in software engineering: A stateof-the-art-report. Fraunhofer Center. Department of Defence.
[54] Happel H. y Maalej W. (2008). Potentials and challenges of recommendation systems for software development. En 2008 international workshop on recommendation systems for software engineering. Atlanta, USA.
[55] Robillard M. et al. (2010). Recommendation systems for software engineering. IEEE Software 27(4), 80–86.
[56] Bajracharya S. y Lopes C. (2009). Mining search topics from a code search engine usage log. En 6th IEEE international working conference on mining software repositories. Washington, USA.

[57] Aguiar A. et al. (2009). Wikis4SE'2009: Wikis for software engineering. En 31st International Conference on Software Engineering. Vancouver, Canada.
[58] Happel H. et al. (2010). Applications of ontologies in collaborative software development. En I. Mistrík et al. (Eds.). Collaborative software engineering (pp. 109-129). Springer.
[59] Serna E. y Serna A. (2016). Model to Development and Management Requirements Engineering MoDeMaRE. En I Simposio de Investigación Uniremington. Medellín, Antioquia.
[60] Fernández M. y Gómez A. (2002). Overview and analysis of methodologies for building ontologies. The Knowledge Engineering Review 17(2), 129-156.
[61] Smith B. (2004). Beyond Concepts: Ontology as Reality Representation. En International Conference on Formal Ontology and Information Systems. Turin, Italy.
[62] Malafsky G. y Newman B. (2009). Organizing knowledge with ontologies and taxonomies. En Techi LLC. Fairfax, USA.
[63] Law A. (2007). Simulation Modeling and Analysis. McGraw-Hill.

CAPÍTULO XV
Desarrollo y gestión de requisitos: Resultados de una revisión de la literatura[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

En la literatura se propone diferentes modelos para gestionar la Ingeniería de Requisitos que presentan ventajas y desventajas al momento de especificar adecuadamente las necesidades del cliente. Se realizó una revisión de la literatura con el objetivo de determinar las buenas prácticas y valorar las características y potencialidades de los modelos más populares. En este capítulo se describe los resultados y se concluye que un modelo para gestionar la Ingeniería de Requisitos debería reunir las buenas prácticas encontradas, y potencializarlas con aportes desde otras áreas del conocimiento, como los métodos formales, la matemática discreta, la lógica, la abstracción, y otras relacionadas. Esto se debe a que los problemas actuales son más complejos y complicados que antes, la seguridad se ha convertido en un asunto clave, las modificaciones son cotidianas, y los tiempos de entrega de los productos software se han acortado drásticamente, a la vez que los presupuestos se deben ajustar a su vida útil.

[1] Presentado en 8th Euro American Conference on Telematics and Information Systems. Cartagena, Colombia. 2016.

INTRODUCCIÓN

La Ingeniería de Requisitos es la fase más importante del proceso de desarrollo de software [1, 2]. Además, típicamente los requisitos son volátiles y es difícil hacerles seguimiento a los cambios, lo que repercute en las demás fases del ciclo de vida, porque incrementa el costo de los proyectos y afecta el calendario establecido [3]. Si la volatilidad de los requisitos se extiende en el tiempo, la probabilidad de éxito del proyecto se reduce drásticamente, y se vuelve complicado y propenso a errores [4]. Por eso es importante desarrollar y gestionar adecuadamente los requisitos, para que las otras fases no sufran contratiempos inesperados. Berry and Lawrence [5] sugieren que el objetivo de la Ingeniería de Requisitos es introducir los principios ingenieriles en la práctica del análisis de los Sistemas de Información, y para lograrlo se debe implantar bajo un proceso sistemático y disciplinado [6].

Debido a que en las diferentes comunidades de investigación se comprende que para el éxito de los proyectos software es importante atender adecuadamente estos procesos, desde hace algún tiempo han propuesto modelos de gestión de requisitos, conformados por actividades estructuradas y repetibles. Esto se hace evidente cuando la mayor parte de las causas de los fracasos se relaciona con esta fase de la Ingeniería del Software [7]. Por otro lado, las fases subsiguientes del desarrollo del producto dependen en gran medida de la Ingeniería de Requisitos [8, 9], entonces, en el proceso se debe cubrir eficientemente todas las actividades involucradas, como la elicitación, la gestión y la documentación del conjunto de necesidades del sistema y del usuario.

Si esta fase no desarrolla adecuadamente el producto puede sufrir contratiempos, como que no se entregue a tiempo, que desborde el presupuesto, que no satisfaga las necesidades de clientes y usuarios, que sea poco fiable, o que presente demasiados errores. Entre el 40% y el 60% de los defectos de un sistema se relacionan con errores cometidos durante la Ingeniería de Requisitos, el 13% de los proyectos fallan debido a requisitos incompletos y el 9% por el rápido cambio en los mismos. Una forma de evitar estos problemas es aplicar buenas prácticas, procesos, herramientas, tecnologías, metodologías y métodos, cuyo objetivo sea entregar un documento de especificación con suficientes criterios de calidad [10].

En consecuencia, identificar a tiempo los problemas y sugerir mejoras en los procesos de esta fase es un principio que puede incrementar el éxito de los proyectos. Desde que Bell y Taylor [11] publicaron su trabajo, se reconoció que la Ingeniería de Requisitos era un proceso ingenieril, y desde entonces se ha presentado diversos enfoques con el objetivo de hacer precisamente eso: *aplicar ingeniería*. Aunque la mayoría se ha centrado en la funcionalidad que se espera del producto, y no en el desarrollo y la gestión, algunos métodos y técnicas se orientan a comprenderlos y a modelarlos como un paso importante hacia la perfección del proceso [12].

Otros se centran en describir y en proponer mejoras para la práctica, llegando a cuestiones clave de naturaleza organizacional, pero no de carácter técnico, como la gestión de documentos y de la incertidumbre [13, 14]. Sin embargo, algunos intentan construir y proponer modelos orientados directamente al desarrollo y la gestión de requisitos, y otros a proponer modelos prescriptivos, en lugar de examinar los descriptivos en la práctica actual [12]; inclusive, otros se han centrado en los requisitos no-funcionales, y aceptan que los mismos influyen transversalmente en los demás.

El objetivo de esta investigación es realizar una revisión de la literatura con el objetivo de presentar un estudio comparativo a las prácticas y procesos que proponen los modelos divulgados, mediante un análisis a los indicadores experimentales y prácticos que las comunidades

relacionadas reportan. Con esta información se estructura un trabajo futuro, con la idea de integrar esas prácticas en un modelo mejorado, complementado con principios de los métodos formales, la matemática discreta y la lógica, para enriquecer los hallazgos de esta investigación. Esta necesidad se descubre a raíz de que los problemas actuales son más complejos y complicados que antes, la seguridad se ha convertido en un asunto clave, las modificaciones son más cotidianas, y los tiempos de entrega de los productos software se han acortado drásticamente, a la vez que los presupuestos se ajustan a su vida útil.

1. MÉTODO

Las metodologías tradicionales para el análisis de la información tienen en cuenta la documentación, al mismo tiempo que al entrenamiento necesario para procesarla, e implican otras áreas de mayor nivel, como el descubrimiento de la información requerida, el análisis, la identificación de posibles soluciones, la adecuada producción y la entrega de resultados [15-17]. Por otro lado, los métodos tradicionales de recolección y análisis de datos incluyen al menos una de estas fases, y frecuentemente utilizan entrevistas estructuradas o técnicas grupales para descubrir los datos necesarios. Posteriormente, los analistas pueden modificar la información estructurada para satisfacer las necesidades de clientes y usuarios, y en este proceso deben tener en cuenta los principios de gestión de la calidad, abarcando los diferentes sistemas, estructuras y actores relacionados [18, 19].

Con este objetivo diferentes empresas e investigadores han propuesto técnicas relacionadas con la documentación computacional, que se utilizan en diferentes escenarios [20]. Es el caso de la encuesta contextual, utilizada para analizar los requisitos de la información, y que se desarrolló como una técnica de investigación interpretativa para recoger y analizar en profundidad los requisitos de usuario, en lo que tiene que ver con el diseño de productos y servicios. El enfoque se basa en un proceso cuidadoso de observación y discusión en el que se involucra a los participantes del proyecto y a posibles miembros del público, y su aplicación varía en función de los resultados y datos deseados. Los requisitos de usuario se recogen mediante la observación y la conversación con especialistas y administradores comprometidos en el problema, pero se obtiene mejores resultados cuando el cliente se involucra más en los procesos de la organización, lo que generalmente implica el trabajo de varias personas o de un equipo.

El mapeo es otra forma de análisis de información que involucra estrategias y técnicas para crearla, incluso cuando el público es variado y difícil de identificar. Es un enfoque fácil de aprender, y tiene por objeto ayudarles a los diseñadores a analizar, organizar y presentar la información y los materiales de capacitación en un formato modular. Antes de usar un método particular para elicitar requisitos, los desarrolladores necesitan comprender los principios básicos, los tipos, los bloques de información y los mapas.

Weiss [21] propone un enfoque estructurado para el análisis de las necesidades de información que una organización requiere para un producto o servicio que, si se aplica adecuadamente, lo elicitado será público, respetará las reglas y será explícito, debido a que se inicia con una imagen lo más amplia posible que posteriormente le permitirá al diseñador agregar superposiciones en etapas sucesivas. El modelo incluye actividades como las habilidades y destrezas del equipo, una lista de características y temas, y el análisis de la audiencia, de la cual crea una matriz y determina los parámetros de los formatos a aplicar.

La Ingeniería de Requisitos, como primera fase del proceso de la Ingeniería del Software, es la tarea más importante para el desarrollo de productos [20, 22], y los requisitos ambiguos son una

de las principales razones por las que fracasan los proyectos [23] y generan defectos en el producto [24, 25]. Debido a esto se reconoce a la Ingeniería de Requisitos como una fase crítica para el desarrollo de software, y actualmente existe diversas metodologías y técnicas para gestionarla. Sin embargo, algunas, que han demostrado éxito para un sistema, parecen no funcionar bien para otro, lo que prueba que la selección de alguna de ellas puede ser difícil, e incluso llevar al fracaso del sistema. Los cambios continuos en los entornos empresariales también afectan el proceso de esta fase, además, cuando el tiempo para la liberación de las versiones es crítico, una inadecuada selección afectará la calidad del producto, por lo que los modelos y metodología deben proporcionar directrices apropiadas para que los ingenieros tengan una buena base para tomar decisiones.

2. RESULTADOS

Los modelos para la Ingeniería de Requisitos deben describir claramente no solo actividades que identifican los requisitos del software y del sistema, sino también el entorno de desarrollo, es decir, metas, puntos de vista, necesidades de las partes interesadas, entre otras. A continuación, se describe y analiza los diferentes modelos para gestionar requisitos que se encuentran en la literatura, y dado que cada uno tiene un enfoque particular, es necesario analizarlos y comprenderlos con la finalidad de reunir la información necesaria para alcanzar el objetivo de la presente investigación.

2.1 NATURE Framework

Este modelo describe al proceso de la Ingeniería de Requisitos como un espacio ortogonal de tres dimensiones, como se observa en la Figura 1, y parte del principio de que al comenzar las etapas el conocimiento del sistema es impreciso, por lo tanto, la especificación es *opaca*, basada en puntos de vista personales, y expresada principalmente mediante representaciones informales. Para que la salida deseada del proceso de especificación de requisitos sea precisa, debe ser *completa*, expresada en un *lenguaje formal*, y *comúnmente aceptada* por todas las partes interesadas, lo que se constituye en los principales objetivos del modelo.

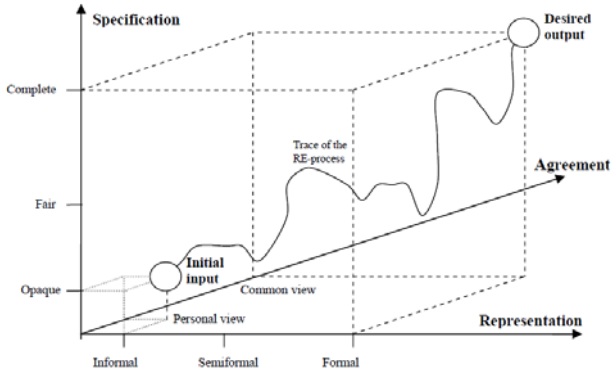

Figura 1. NATURE framework [26, 27]

1. La dimensión de *representación* de requisitos aborda el grado de formalidad de la representación, y tiene el objetivo de transformar al modelo informal en un modelo formal de requisitos.
2. La dimensión de *contrato* de requisitos trata el grado del acuerdo de requisitos, y su objetivo es obtener un acuerdo común acerca del modelo.
3. La dimensión de *especificación* de requisitos se encarga del grado de comprensión de los mismos, en un momento determinado de tiempo. Su objetivo es mejorar la comprensión del

sistema, hasta entonces opaca, mediante una especificación completa de requisitos, en la que la integridad se mide mediante estándares y directrices.

2.2 Iterative Requirements Engineering Process Model

Es un modelo no-lineal conformado por tres actividades: elicitación, especificación y validación, que representan el proceso de la Ingeniería de Requisitos como iterativo y cíclico por naturaleza, como se observa en la Figura 2. Además, demuestra las interacciones entre la elicitación, la especificación, la Validación, el usuario y el dominio del problema. Aunque contiene actividades similares a las que proponen los lineales, el orden en el que aparecen no lo es, y sugiere una relación de causa y efecto entre ellas.

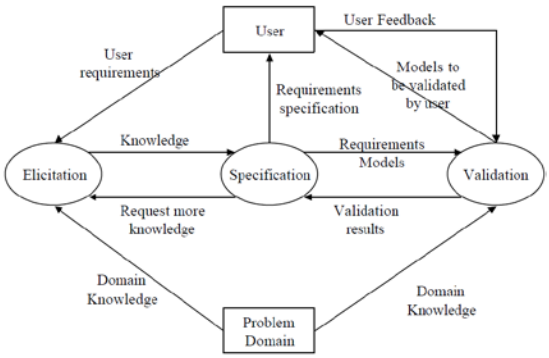

Figura 2. Iterative Requirements Engineering Process Model [28]

1. En la *elicitación* se adquiere el conocimiento necesario para producir un modelo formal de requisitos.
2. En la *especificación* se reciben como entrada los entregables de la elicitación, y el objetivo es crear un modelo formal de requisitos. La especificación de requisitos se puede ver aquí como un documento que define la funcionalidad deseada, sin mostrar cómo se va a alcanzar [29].
3. La *Validación* certifica que el modelo formal de requisitos producido satisface las necesidades de los usuarios.

Este modelo se centra en el principio del desarrollo *inhouse*, porque el sistema se desarrolla al interior de la misma empresa que lo requiere, y en los proyectos de *desarrollo por contrato*, cuando un proveedor desarrolla un sistema para una empresa cliente [30]. En este tipo de proyectos la especificación y sus salidas se definen como un contrato entre clientes y desarrolladores, sin embargo, el modelo no se ocupa de proyectos en los que no está definido el cliente, por ejemplo, en el desarrollo comercial *off-the-shelf*.

2.3 Purely linear process model

Se trata de un modelo puramente lineal, como se observa en la Figura 3, que no tiene en cuenta la superposición o iteración de actividades, como el conceptual lineal, sino que las clasifica en diferentes grados, aunque la progresión lineal resultante en la documentación es común a ambos modelos. Por otro lado, reconoce que los procesos de la Ingeniería de Requisitos dependen de la situación, y analiza siete diferentes relaciones entre el cliente y el proveedor, y sus correspondientes procesos en esta fase. El modelo está conformado por cinco actividades organizadas secuencialmente, y debido a su simpleza se utiliza principalmente en pequeños proyectos con nivel de complejidad bajo, pero no es adecuado ni recomendable para proyectos grandes y complejos.

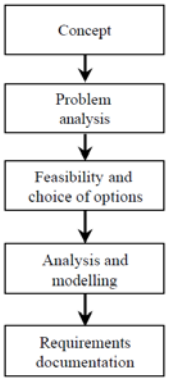

Figura 3. Purely linear process model [31]

2.4 The PREview process

PREview es un enfoque orientado a puntos de vista VOA, que considera diferentes perspectivas a la información relacionada con el problema, llamados puntos de vista, que surgen debido a las diferentes responsabilidades, funciones, objetivos, ... de las fuentes de información (Figura 4).

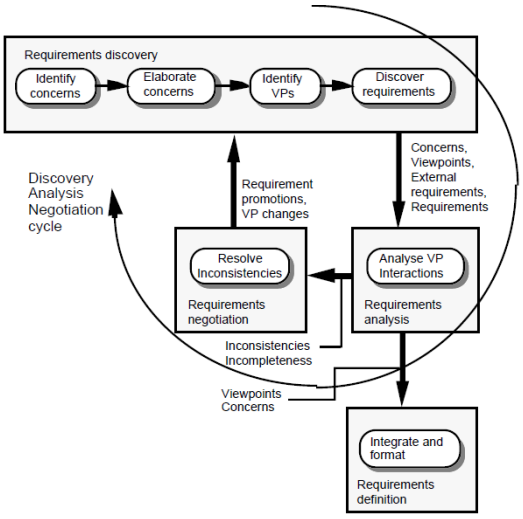

Figura 4. The PREview process [32]

Este enfoque complementa la noción estándar de los puntos de vista con el de las cuestiones organizacionales, como la generalización de la noción de objetivo, que incluye tanto los objetivos como las limitaciones organizacionales que restringen el sistema o proceso a analizar. La fortaleza de PREview se sustenta en el énfasis en las necesidades de las partes interesadas, y la facilidad de integración con los enfoques de requisitos. Además, proporciona un adecuado punto de partida para que los usuarios identifiquen los puntos de vista.

2.5 Conceptual linear process model

A diferencia de los modelos lineales e incrementales, en los que las actividades de elicitación y análisis se combinan bajo diferentes premisas, pero siguiendo una transición lineal similar, los autores proponen un modelo conceptual lineal en el que indican iteraciones entre actividades que se solapan, y que a menudo se realizan iterativamente, como se observa en la Figura 5.

En este tipo de modelos las actividades se repiten en iteraciones, formando una espiral, y al final de cada una se toma una decisión de si se debe aceptar el documento construido o realizar iteraciones adicionales. Para lograrlo, el modelo separa los procesos de la fase en dos etapas: 1)

actividades propiamente dichas, y 2) gestión de requisitos, que comprenden los principios de formalidad, concordancia y comprensibilidad de un modelo de requisitos. La primera está conformada por la elicitación, el análisis y negociación, la documentación y la Validación de requisitos. En paralelo con estas actividades se desarrolla el proceso de gestión de requisitos, que se encarga de la gestión de los cambios que se presentan en la evolución de los mismos. Estos cambios son necesarios, porque a medida que aparecen nuevas necesidades permiten el descubrimiento de errores en los requisitos. Esta etapa debe hacerles un seguimiento constante a los cambios, asegurándose de que se realicen de manera controlada.

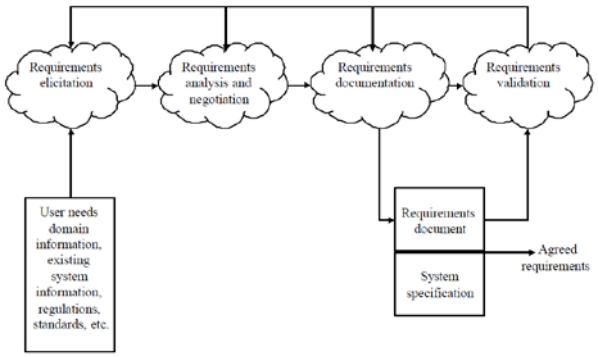

Figura 5. Conceptual Linear Requirements Engineering Process Model [33]

2.6 Extreme Programming XP

Al igual que RUP, XP tiene un carácter iterativo e incremental (Figura 6) y, a primera vista, XP y un análisis de requisitos fundamental parecen contradecirse entre sí, porque el primero está orientado a conseguir un sistema rápidamente implementable en el mercado [34, 35]. Sin embargo, en este modelo también hay enfoques que se complementan bien en el análisis de requisitos, como las historias de usuarios, el juego de planificación y la metáfora del sistema. Por ejemplo, las historias de usuario son informes breves que realizan los usuarios, los cuales se reúnen inicialmente de manera informal; posteriormente, se añade los detalles poco a poco, y a continuación se evalúan.

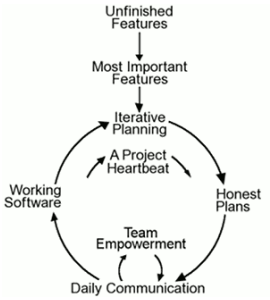

Figura 6. Xtreme Programming model [34]

2.7 Problem Frames

Este modelo propone descomponer los problemas complejos en conjuntos estructurados de clases simples y sub-problemas comunes familiares, como se observa en la Figura 7. Las clases de problemas comunes se identifican a partir del análisis del problema, de forma similar al diseño de patrones [36]. Las descripciones/soluciones combinadas para los sub-problemas sirven como descripción/solución para el problema original.

El modelo sugiere que una vez que los desarrolladores conocen el contexto del problema es más fácil reconocer y abordar las variaciones de sus clases, y de anticipar las dificultades, así como

producir soluciones eficientes. A menudo, los tipos de problemas tratados son llamados requisitos funcionales, sin embargo, el método reconoce la importancia de los aspectos no-funcionales y se centra en la composición de las necesidades, para lo que combina contextos de problemas sencillos en marcos compuestos que representan el análisis realista de los problemas complejos.

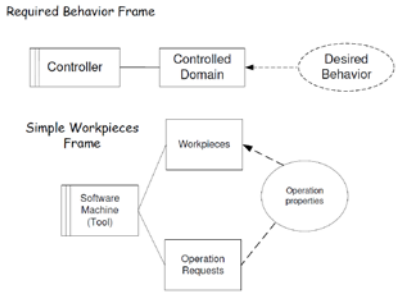

Figura 7. Jackson's problem frames [37]

2.8 Development/Management Model

Esta propuesta separa la Ingeniería de Requisitos en dos grupos de actividades: 1) el proceso o desarrollo de requisitos, y 2) la gestión de requisitos, como se observa en la Figura 8.

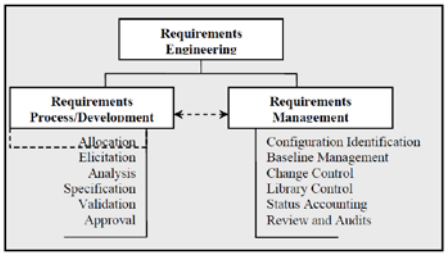

Figura 8. Requirements Development/Management Model [38]

Las actividades del primero grupo tienen como objetivo asignar responsabilidades, involucrar a las partes interesadas en el proceso, y dividir los requisitos en grupos lógicos. Mientras que las del segundo se orientan a gestionar los requisitos mediante una serie de etapas. La aprobación de requisitos se utiliza para determinar si es rentable continuar con el modelo, o si lo mejor es re-conceptualizarlo y aplicar solo una parte del mismo.

2.9 AORE model

Se puede considerar como un modelo general de procesos para la Ingeniería de Requisitos, cuyo objetivo es separar los aspectuales de los no-aspectuales, así como de sus reglas de composición, como se observa en la Figura 9. También describe una instanciación concreta utilizando puntos de vista y un lenguaje de composición basado en XML, junto con una herramienta de soporte llamada Arcade. En el enfoque Arcade, los requisitos aspectuales son similares a los propuestos por PREview, en el que los puntos de vista son transversales y se utilizan para elicitarlos. Ambos modelos están representados mediante un *framework* semi-estructurado basado en XML, que también se utiliza para definir las reglas de composición que emplean acciones informales y las operaciones que reflejan cómo los requisitos aspectuales afectan a los grupos de requisitos transversales.

Las contribuciones más valiosas y novedosas de este enfoque son: su capacidad para separar y luego componer requisitos transversales y no-transversales, un conjunto flexible de operadores de composición, y una presentación original de requisitos en XML. También proporciona un

procedimiento exhaustivo para identificar y resolver conflictos. El apoyo de Arcade para el mapeo de requisitos, aunque muy útil, carece del rigor y la minuciosidad de PREview, o el análisis de NFR.

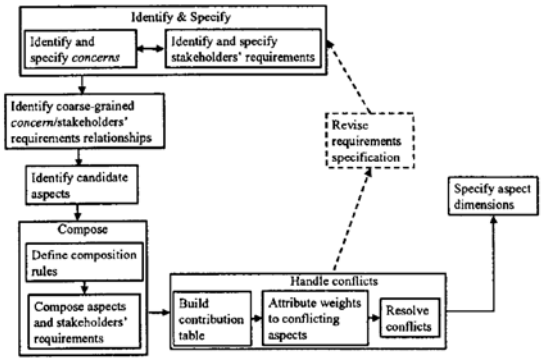

Figura 9. AORE Model [39]

2.10 Linear Iterative Requirements Engineering Process Model

El modelo consta de tres fases principales: 1) elicitación, 2) especificación, y 3) validación de requisitos, pero debido a su naturaleza iterativa generalmente se realizan varias veces hasta la resolución, como se muestra en la Figura 10.

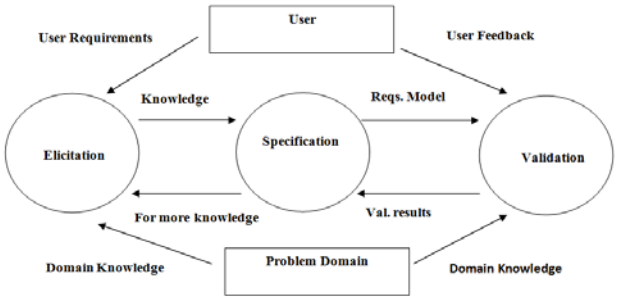

Figura 10. Linear Iterative Requirements Engineering Process Model [40]

Este tipo de modelo de procesos es típico para el desarrollo de software, donde continuamente se produce y libera nuevas revisiones del producto, requiriendo procesos y periodos más cortos. Además, el modelo incorpora comentarios e información de los usuarios y, por tanto, los problemas tienen una mejor posibilidad de ser evaluados antes de la validación. La principal diferencia entre esta propuesta y el modelo completamente lineal es que la fase de validación es iterativa, lo que significa que se puede realizar tantas veces como sea necesario, hasta que los requisitos sean validados por las partes interesadas y acuerden las especificaciones finales del sistema. Este proceso asegura una mejor validación de requisitos.

2.11 Requirements Abstraction Model RAM

Este modelo (Figura 11) se estructura en tres pasos básicos: 1) *Especificar*, que implica especificar el requisito inicial y elicitar información suficiente al respecto para definir un número de atributos; 2) *Ubicar*, que se centra en averiguar el nivel de abstracción en el que se encuentran los requisitos especificados, y para esto propone los siguientes: nivel del producto, nivel de las características, nivel de las funciones y nivel de los componentes; 3) *Abstraer*, en el que se hace un trabajo de depuración de cada requisito, e implica abstraer y/o desglosar los requisitos, dependiendo de su ubicación en el paso anterior. La depuración consiste en crear nuevos requisitos, llamados requisitos *depurados*, sobre los niveles de abstracción adyacentes, o *de vinculación* en función de la situación existente.

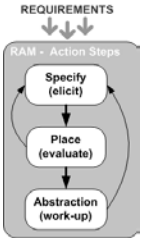

Figura 11. Requirements Abstraction Model RAM [41]

2.12 V Model

El Modelo V es una adaptación del Modelo Cascada utilizado en la ingeniería de sistemas y de software, pero se diferencia principalmente en la gestión de las actividades de validación, pruebas, calidad y riesgos, que son manejadas mediante la adición de varias fases iterativas a la estructura del proceso. La intención es llevar a cabo procedimientos de prueba y verificación de forma continua, para mejorar la calidad y minimizar los riesgos, como se observa en la Figura 12.

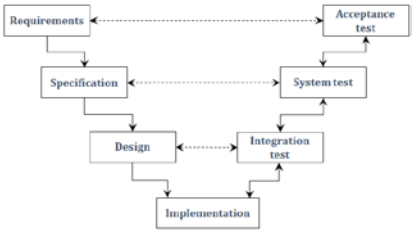

Figura 12. Model V [42]

Las fases de desarrollo pueden ser vistas como diferentes capas que progresan desde la etapa inicial de alto nivel, bajando en el proceso hasta que alcancen los niveles más bajos, donde se puede iniciar el trabajo de ejecución. En cada fase se re-validan los requisitos especificados y las fases del modelo pueden variar en función de las intenciones y el proyecto, pero el patrón básico debe permanecer igual.

2.13 Rational Unified Process RUP

RUP es un modelo de proceso de desarrollo de software que consiste de dos dimensiones de procesos (Figura 13): 1) de tiempo, que indica una subdivisión en una estructura en bruto (fases) y una estructura refinada (iteraciones), y 2) se refiere a la parte técnica, y la divide en disciplinas, de las cuales hay seis de proceso primario (incluyendo requisitos) y tres de infraestructura; además, cada disciplina tiene su propio flujo de trabajo definido.

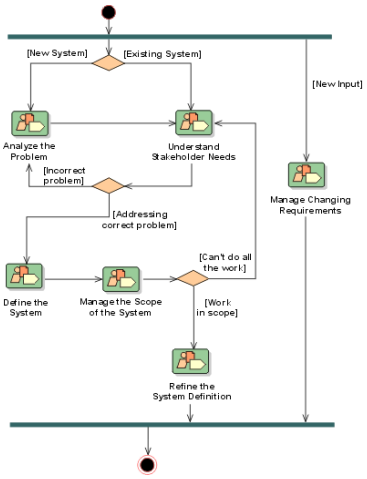

Figura 13. RUP Requirements Discipline [43]

La Ingeniería de Requisitos tiene como objetivo la fiabilidad de las especificaciones y el desarrollo, así como las modificaciones al sistema. En esencia, en RUP se compone de las siguientes actividades principales: 1) análisis del problema, 2) comprensión de las necesidades de las partes interesadas, 3) definición del sistema, 4) gestión del alcance del sistema, 5) cambios en los requisitos, y 6) refinamiento de la definición del sistema. Estas actividades están lógicamente conectadas entre sí y no se deben considerar como puramente secuenciales.

2.14 Requirements Methodology for Agent Oriented Paradigm

La Ingeniería de Requisitos implica los procesos de adquisición y de establecimiento de los requisitos finales de un sistema, y cuando se orienta a agentes modela los requisitos de un sistema en términos de las tareas y metas que varios agentes pueden exhibir. El enfoque orientado a agentes que propone este modelo (Figura 14) se basa en un modelo en espiral, para dar cabida a la Ingeniería de Requisitos dinámica, que sugiere una versión limitada de requisitos para satisfacer las demandas competitivas o de negocios. Pero los detalles de los requisitos exigen un modelo de proceso para dar cabida a un requisito de producto que evoluciona con el tiempo.

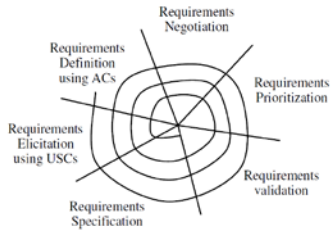

Figura 14. Requirements Methodology for Agent Oriented Paradigm [44]

Cada ronda del ciclo captura requisitos adicionales y le facilita al desarrollador negociar, priorizar y validar los requisitos encapsulados. El proceso comienza con la elicitación de los requisitos en forma de historias de usuario. La información acumulada en User Story Card USC se elabora mejor en forma de Agent Card AC, para promover la comprensión del desarrollador de la viabilidad de los requisitos orientados a agentes. Los requisitos ampliados almacenados en las AC se priorizan con base en negociaciones, que se llevan a cabo entre las partes interesadas.

2.15 Goal Elicitation Method

Como se observa en la Figura 15, este método básicamente comprende dos fases consecutivas: 1) elicitación preliminar, para recoger una primera versión del modelo, y 2) elicitación con catálogos, para complementar y refinar los modelos previamente derivados por medio de requisitos no-funcionales.

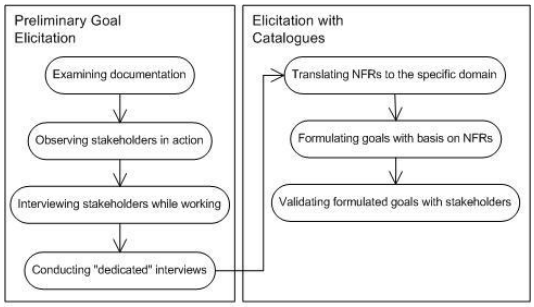

Figura 15. Goal Elicitation Method [45]

De acuerdo con la fuente de información y las técnicas utilizadas para interactuar con los actores del proceso, la primera fase se divide en cuatro etapas: 1) se examina la documentación acerca

de los procesos organizacionales; 2) se obtiene un modelo preliminar de objetivos, junto con un modelo de procesos de negocio preliminar; 3) mientras se observan sus acciones se centra en la elicitación de requisitos mediante entrevistas a los actores de la organización; y 4) se concentra en entrevistas dedicadas, no solo a los actores de procesos de negocio, sino también con los jefes de departamento. En la segunda fase se emplea el marco NFR [46-48], con el objetivo de abstraer y extrapolar una serie de requisitos no-funcionales, para identificar objetivos que tienen relevancia estratégica para los modelos de procesos de negocio y que no habían sido identificados previamente.

2.16 Volere

El modelo Volere fue desarrollado especialmente para la Ingeniería de Requisitos y, además de técnicas para determinar requisitos, proporciona plantillas para estructurar la especificación. Está organizado como se observa en la Figura 16 para proporcionales a los usuarios una plantilla de Ingeniería de Requisitos sistemática, estructurada y completa. Toda la información en ella se mantiene en un documento (monolítico), a la inversa de RUP. Además, con el fin de desarrollar los requisitos utiliza otra plantilla de requisitos. El aseguramiento de calidad es un paso intermedio, denominado, que se utiliza entre la especificación y el análisis de requisitos. El modelo también se debe considerar iterativo.

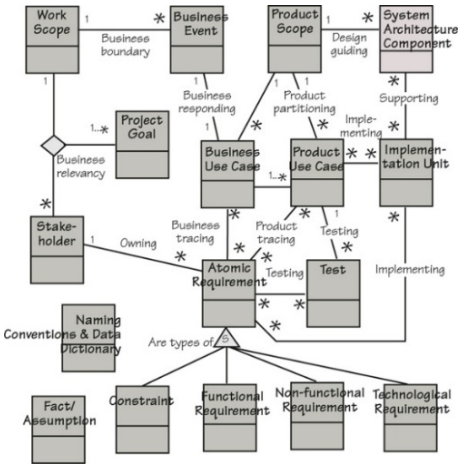

Figura 16. Volere model [49]

3. ANÁLISIS DE RESULTADOS

La Ingeniería de Requisitos abarca varias actividades importantes del ciclo de vida de la Ingeniería del Software, como la elicitación, el análisis, la especificación, la resolución de conflictos y la validación. El objetivo principal de esta fase es especificar claramente las necesidades del cliente, de tal forma que los ingenieros de software obtengan un amplio conocimiento de las funciones, restricciones y propiedades del sistema a desarrollar, así como del entorno en el que funcionará.

La comunidad de investigadores ha desarrollado y propuesto diversos modelos y enfoques para lograr este objetivo, y a pesar de que han sido parcialmente aplicados y validados, presentan inconvenientes, como que se basan en ejemplos relativamente simples, o que se llevan a cabo de forma aislada. De hecho, no reflejan un estudio empírico realizado en el campo que estudie de manera sistemática cómo evaluar y comparar la eficacia y eficiencia de los modelos. Además, tampoco ofrecen evidencias empíricas de las ventajas e inconvenientes de la práctica actual de la Ingeniería de Requisitos, por lo que no se basan en un marco de evaluación estructurado, lo que impide su desarrollo, ejecución y reproducción en otros casos de estudio.

En este trabajo se presenta un análisis a varios de los modelos divulgados, y la conclusión es que son diferentes, y a veces de naturaleza conflictiva, que van desde estructuras lineales e incrementales a cíclicas y reiterativas. Los resultados de estudios previos indican que en la práctica algunos difieren de lo comúnmente aceptado en la literatura [50, 51]. Además, esta complicación es situacional y se ve afectada por la relación cliente-proveedor [31], por el producto, por la madurez técnica, y por la participación disciplinar y la cultura de la organización [33].

Aunque los procedimientos y la documentación de la mayoría de los modelos propuestos son claros y bien presentados, las validaciones experimentales relacionadas demuestran que el seguimiento paso a paso no es suficiente para gestionar adecuadamente la Ingeniería de Requisitos. Algunos alcanzan la especificación suficiente, pero no necesaria, para comprender adecuadamente el problema; otros no documentan el proceso y generan incomprensión de los requisitos; otros no estructuran el diálogo entre las partes, por lo que se debe reformular continuamente el proceso de elicitación. En general se puede concluir que ninguno de los modelos analizados es suficiente para gestionar adecuadamente y de forma estructurada la Ingeniería de Requisitos, y que se necesita integrar las mejores prácticas de varios de ellos, complementadas con principios desde otras áreas del conocimiento, para proponer un modelo general de gestión y administración de esta fase.

Un nuevo modelo debería reunir las buenas prácticas encontradas, y potencializarlas con aportes desde otras áreas del conocimiento, como los métodos formales, la matemática discreta, la lógica, la abstracción, y otras relacionadas. Esto se debe a que los problemas actuales son más complejos y complicados que antes, la seguridad se ha convertido en un asunto clave, las modificaciones son más cotidianas, y los tiempos de entrega de los productos software se han acortado drásticamente, a la vez que los presupuestos se ajustan a su vida útil.

4. CONCLUSIONES

Las buenas prácticas encontradas en esta revisión se pueden utilizar en muchos aspectos en beneficio de un modelo integral para desarrollar y gestionar la Ingeniería de Requisitos. Valdría la pena realizar otras investigaciones para identificar un mayor número de buenas prácticas en otros modelos que no fueron cubiertos en esta revisión, o que no han sido publicados desde la industria, aunque de uso cotidiano. Aunque muchas buenas prácticas han sido identificadas en trabajos anteriores, todavía hay que determinar si podrían ser desplegadas en un modelo integral, porque se debe cumplir una serie de condiciones necesarias, antes que puedan aplicarse con éxito y ayudarles a los ingenieros de requisitos a realizar de mejor manera su labor.

Mientras se pueda identificar un mayor número de buenas prácticas se podrá descubrir o estructurar modelos más inteligentes, y pensar en la implementación de herramientas de automatización a través de la integración de niveles más altos de formalidad. Porque, actualmente, y como se puede observar en la presente revisión, las propuestas contienen muchas reglas que se deben ejecutar para lograr una especificación adecuada, y aunque la mayoría proporciona soporte inteligente, su rendimiento decae significativamente al momento de aplicarlos. En tal caso se requiere mayor nivel de integración y de apropiación de las reglas inteligentes, se necesita más análisis de gestión, un mejor conocimiento del dominio de los requisitos, mejorar el diseño y la documentación de los requisitos elicitados, y conocer adecuadamente los hábitos de los usuarios y demás partes interesadas.

Un modelo para desarrollar y gestionar la Ingeniería de Requisitos debería reunir las buenas prácticas encontradas en esta revisión, y potencializarlas con aportes desde otras áreas del

conocimiento, como los métodos formales, la matemática discreta, la lógica, la abstracción, y otras relacionadas. Además, el modelo se debe estructurar para hacerles frente a las debilidades encontradas en los propuestos hasta el momento, porque todavía no brindan un soporte inteligente para la Ingeniería de Requisitos, y aún no cubren todas las cuestiones en el desarrollo y gestión de los mismos, sin embargo, los resultados iniciales son prometedores. En términos generales se podría concluir que:

- Un modelo de Ingeniería de requisitos debe definir actividades que tengan como objetivo identificar requisitos, proporcionando otras de soporte que contribuyan al éxito y a la calidad de los mismos.
- Para los ingenieros, la flexibilidad en la implementación y la integración de los métodos existentes influyen en el éxito del enfoque que adoptan. La ausencia de asociación para notaciones y métodos particulares es un factor clave en la aceptación del enfoque.
- La introducción de nuevos conceptos puede causar problemas, pero una sesión introductoria, para que los usuarios se familiaricen con los conceptos, puede contribuir al éxito de la implementación del concepto.
- Se necesita un enfoque iterativo en Ingeniería de Requisitos para que estos evolucionen. Sin embargo, se requiere una guía para determinar cuándo comienza o termina cada etapa de esta fase.
- Los ingenieros de requisitos deben buscar activamente los requisitos, en lugar de enfatizar en la elicitación desde fuentes conocidas. La elicitación puede ocurrir cuando sea posible identificar la fuente.

REFERENCIAS

[1] Serna E. (2012). Analysis and Selection to Requirements Elicitation Techniques. En 7th Colombian Computing Congress. Medellín, Antioquia.

[2] Silva R. y Dasilva A. (2012). Elements of a Requirements Management Tool to Improve Software Development. RACCIS 2(1), 37-42.

[3] Boehm B. y Papaccio P. (1988). Understanding and Controlling Software Costs. IEEE Transactions on Software Engineering 14 (10), 1462-1477.

[4] Brooks F. (1995). The Mythical Man Month. Addison Wesley.

[5] Berry D. y Lawrence B. (1998). Requirements Engineering. IEEE Software 15(2), 26-29.

[6] Leite J. (1987). A Survey on Requirements Analysis. Advanced Software Engineering Project. Technical Report RTP-071. University of California at Irvine.

[7] Pfleeger S. (2010). Software Engineering – Theory and Practice. Prentice-Hall.

[8] Stevens R. et al. (1998). Systems Engineering: Coping with Complexity. Prentice-Hall.

[9] Saiediana H. y Daleb R. (2000). Requirements engineering: Making the connection between the software, developer and customer. Information and Software Technology 42(6), 419-428.

[10] Drehmer D. y Dekleva S. (2001). A note on the evolution of Software Engineering Practices. Journal of Systems and Software 57(1), 1-7.

[11] Bell T. y Taylor T. (1976). Software Requirements: Are they Really a Problem? En International Conference on Software Engineering. San Francisco, USA.

[12] Madhavji N. et al. (1994). Elicit: A Method for Eliciting Process Models. En 1994 CAS Conference. Toronto, Canada.

[13] Lubars M. et al. (1993). A Review of the State of the Practice in Requirements Modeling. En IEEE International Symposium on Requirements Engineering. San Diego, USA.

[14] El Emam K. y Madhavji N. (1995). A Field Study of Requirements Engineering Practices in Information Systems Development. En Second International Symposium on Requirements Engineering. York, England.

[15] Siddiqi J. (1996). Requirement Engineering: The Emerging Wisdom. IEEE Software 13(2), 15-19.
[16] Mcquilten G. (2007). MIS-design: Art in a consumer landscape. Doctoral Dissertation. Melbourne University.
[17] Caldwell J. (2009). Approach to Management Information System Design. Recuperado: http://www.foundationwebsite.org/ApproachToMIS.htm
[18] Day K. y Devlin R. (1998). The Payoff to Work without Pay: Volunteer Work as an Investment in Human Capital. The Canadian Journal of Economics 31(5), 1179-1191.
[19] Jalote P. (2005). An Integrated Approach to Software Engineering. Narosa Publishing House.
[20] Asghar S. y Umar M. (2010). Requirement Engineering Challenges in Development of Software Applications and Selection of Customer-off-the-Shelf (COTS) Components. International Journal of Software Engineering 1(2), 32-50.
[21] Weiss R. (1997). Strategic Management and Information Systems: An Integrated Approach. Financial Times.
[22] Wahono R. (2003). Analyzing Requirements Engineering Problems. En IECI Japan Workshop. Chofu Bunka Kaikan Tazukuri, Japan.
[23] Hofmann H. y Lehner F. (2001). Requirements Engineering as a Success Factor in Software Projects. IEEE Software 18(4), 58-66.
[24] Young R. (2001). Effective Requirements Practices. Addison-Wesley.
[25] Fernandez D. et al. (2011). A case study on the application of an artefact-based requirements engineering approach. En 15th International Conference on Evaluation and Assessment in Software Engineering. Durham University, UK.
[26] Pohl K. (1994). The Three Dimensions of Requirements Engineering a Framework and its Applications. Information systems 19(3), 243-258.
[27] Pohl K. (1996). Process Centred Requirements Engineering. John Wiley.
[28] Loucopoulos P. y Karakostas V. (1995). System Requirements Engineering. McGraw-Hill.
[29] Davis A. (1993). Software Requirements - Objects, Functions and States. Prentice-Hall.
[30] Lauesen S. (2002). Software Requirements - Styles and Techniques. Addison-Wesley.
[31] Macaulay L. (1996). Requirements Engineering. Springer.
[32] Sawyer P. et al. (1996). PREview: Tackling the Real Concerns of Requirements Engineering. Technical Report: CSEG/5/1996. Lancaster University.
[33] Kotonya G. y Sommerville I. (1998). Requirements Engineering – Processes and Techniques. Wiley.
[34] Beck H. (2000). Extreme Programming Explained. Addison-Wesley.
[35] Balzert H. (2008). Lehrbuch der Software technik: Software management. Spektrum Akademischer.
[36] Gamma E. et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
[37] Jackson M. (2001). Problem Frames: Analyzing and Structuring Software Development Problems. ACM Press.
[38] Ferdinandi P. (2002). A Requirements Pattern. Succeeding in the Internet Economy. Addison-Wesley.
[39] Rashid A. et al. (2003). Modularisation and Composition of Aspectual Requirements. En 2nd International Conference on Aspect Oriented Software Development. Boston, USA.
[40] Kotonya G. y Sommervile I. (2003). Requirements Engineering process and techniques. Wiley.
[41] Gorschek T. y Wohlin C. (2006). Requirements Abstraction Model. Requ. Engin. Journal 11(1), 79-101.
[42] Rupp C. (2006). Requirements-Engineering and –Management. Hanser Fachbuchverlag.
[43] Passing J. (2007). Requirements Engineering in the Rational Unified Process. En Seminar Requirements Engineering. Hasso Plattner Insitute for Software Systems Engineering, Germany.
[44] Gaur V. et al. (2010). An Agent-Oriented Approach to Requirements Engineering. En 2nd International Advance Computing Conference. London, UK.
[45] Cardoso E. et al. (2011). A Method for Eliciting Goals for Business Process Models based on Non-Functional Requirements Catalogues. Inter. Jour. of Inform. System Modeling and Design 2(2), 1-18.
[46] Chung L. et al. (2000). Non-Functional Requirements in Software Engineering. Kluwer Academic.
[47] Cysneiros L. (2007). Evaluating the Effectiveness of using Catalogues to Elicit Non-Functional Requirements. En 10th Workshop in Requirements Engineering. Toronto, Canada.
[48] Lamsweerde A. (2000). Requirements Engineering in the Year 00: A Research. En 22nd International Conference on Software Engineering. Limerick, Ireland.

[49] Robertson S. y Robertson J. (2012). Mastering the Requirements Process: Getting Requirements Right. Addison-Wesley.
[50] Nguyen L. y Swatmann P. (2000). Managing the Requirements Engineering Process. Deakin University.
[51] Houdek F. y Pohl K. (2000). Analyzing requirements engineering processes: A case study. En 11th International Workshop on Database and Expert Systems Applications. Greenwich, UK.

CAPÍTULO XVI
Modelo para Desarrollar y Gestionar la Ingeniería de Requisitos

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

El proceso de desarrollo de software es un ejercicio complejo en el que las características del producto se consideran desde diferentes puntos de vista, roles, responsabilidades y objetivos. Esto les exige a los ingenieros de software aplicar prácticas estructuradas en cada una de las fases del ciclo de vida del producto. La Ingeniería de Requisitos es la fase más importante de este proceso y su objetivo es elicitar, analizar, gestionar y especificar las necesidades de los clientes y usuarios, para luego generar un documento que se convierta en guía para las demás fases. En este trabajo se propone el Modelo para Desarrollar y Gestionar la Ingeniería de Requisitos MoDeMaRe, con el que es posible especificarlos estructuradamente para las sub-siguientes fases del ciclo de vida. En él se integra, mediante procesos iterativos, las buenas prácticas de la investigación en el área y los principios de la lógica, la abstracción y los métodos formales para generar un documento de especificación con criterios de calidad adecuados.

INTRODUCCIÓN

Los requisitos son atributos de los productos software que se deben *descubrir* antes de iniciar su desarrollo. Son una condición o capacidad que debe cumplir o poseer un sistema para satisfacer un contrato, una norma, una especificación u otros documentos estructurados formalmente [1]. Un requisito bien especificado es una afirmación de que la funcionalidad del sistema satisface adecuadamente las necesidades, y actualmente es posible identificar una relación recíproca entre seres humanos y máquinas para elicitar requisitos, que colaboran para desarrollar productos de calidad [2]. Normalmente, los requisitos se clasifican como *funcionales* y *no-funcionales*. Los primeros especifican acciones implícitas que realiza un sistema sin tener en cuenta sus limitaciones físicas, y los segundos especifican propiedades explicitas del producto, tales como limitaciones relacionadas con el medio ambiente, la implementación, el rendimiento, las dependencias de plataforma, la facilidad de mantenimiento, la capacidad de extensión y la confiabilidad, entre otras [1].

La Ingeniería de Requisitos es un enfoque sistemático a través del cual se elicita las necesidades de las diferentes partes interesadas y se documentan para el resto del ciclo de vida. Es un proceso iterativo en el que la gestión es un aspecto de alta importancia [3-5], porque los productos de esta fase son esenciales para las demás actividades del desarrollo. Tradicionalmente, se realiza como una fase temprana del desarrollo de productos software [6], sin embargo, en la práctica especificar un conjunto preciso de requisitos, y mantenerlo estable a lo largo de la línea de tiempo, es una tarea casi *imposible* [7]. Por lo tanto, esta ingeniería se debe concebir como un proceso incremental e iterativo, que se ejecuta y enriquece paralelamente con las otras actividades del desarrollo, tales como el modelado, el diseño, la arquitectura, la codificación y la implementación.

Como tal, la fase es el proceso más crítico y complejo en el desarrollo de productos software [1, 8, 9], porque el documento de especificación será la guía para las demás fases del ciclo de vida, y para describir la transdisciplinariedad de sus procesos y patrones en las diversas interacciones sociales que tienen lugar. Además, tiene un impacto predominante en la funcionalidad y en la calidad del producto final; en ella confluye un conjunto diverso de solicitudes proveniente desde diversos actores, que el equipo de desarrollo debe considerar. Por estas razones se considera como una fase compleja a la vez que crítica que requiere gestión y administración adecuadas.

Los resultados de los estudios experimentales de los procesos de la Ingeniería de Requisitos indican que los modelos sistemáticos e incrementales, que se encuentran en la literatura, no siempre coinciden con las necesidades de los procesos en la práctica. Martin et al. [10] los examinó en un caso de estudio y encontraron que generalmente los proyectos son manejados siguiendo un modelo lineal, con muy poca repetición de actividades. La mayoría de los proyectos que examinaron siguen un proceso directo hasta la fase de creación de prototipos, y solamente entonces dan lugar a un proceso iterativo.

Estos autores indican que el modelo de Loucopoulos y Karakostas [11] es una buena representación del proceso *ad hoc* y de la naturaleza iterativa del prototipado, pero no muestra la progresión de las fases. Nguyen y Swatman [12] también utilizaron un estudio de caso y encontraron que el proceso de la Ingeniería de Requisitos no ocurre de forma sistemática, fluida e incremental. Más bien es oportunista, con simplificación esporádica, que re-estructura el modelo de requisitos cuando encuentra puntos de alta complejidad. Houdek y Pohl [13] también realizaron un estudio de caso, pero no pudieron reproducir un modelo monolítico para las actividades de Ingeniería de Requisitos, porque estaban demasiado entrelazadas y porque tradicionalmente se ven como tareas separadas.

Los estudios de campo en Ingeniería de Requisitos también han reunido resultados contradictorios en cuanto la efectividad de los procesos en las organizaciones. Lo que indica que esta área no ha madurado completamente, en el sentido que no se ha podido determinar un modelo utilizado y aceptado universalmente. Kotonya y Sommerville [4] afirman que son pocas las empresas que tienen una definición estándar de estos procesos. En consonancia con esto, Hofmann y Lehner [14] examinaron los procesos de Ingeniería de Requisitos que se aplican en la industria, y encontraron que la mayoría la realizan como una actividad *ad hoc*, resultados que también confirman Lowe y Eklund [15] en sus estudios.

En contraste con estos resultados, Emam y Madhavji [16] llegaron a la conclusión de que las organizaciones tienden a utilizar procesos estándar, porque los consideran como las mejores prácticas. Chatzoglou [17] aplica una encuesta en línea para examinar el proceso de la Ingeniería de Requisitos, con el objetivo de entender las diferencias entre proyectos con diversas características, pero haciendo especial hincapié en los recursos humanos. Las principales conclusiones de este estudio es que se debe utilizar una metodología de proceso estándar, que debe adaptarse a las necesidades específicas de cada proyecto, y que los recursos deben ser puestos en la primera iteración del proceso de la Ingeniería de Requisitos. Pero de los 64 proyectos evaluados encontró que solamente el 10% lo tiene parcialmente en cuenta.

En términos generales la Ingeniería de Requisitos se conforma de actividades como elicitar, analizar, gestionar, documentar, validar y mantener el conjunto de requisitos de un sistema determinado [18, 19]. Todos estos procesos son necesarios para el éxito de los proyectos software, y no mejorarlos o hacerlo de forma no-estándar puede desmejorar posteriormente las posibilidades de lograr el éxito en el desarrollo. Antes de elaborar y proponer modelos y estrategias para esos procesos, es necesario investigar y analizar los que se han propuesto y publicado, para construir un cuerpo de conocimiento sólido que los sustente. Ese ha sido el objetivo del trabajo del profesor Serna [20, 21], en el que realiza una investigación mediante revisión de la literatura para determinar si es posible encontrar un modelo aceptado ampliamente para gestionar esta fase del ciclo de vida. Los hallazgos demuestran que no es posible debido a la diversidad de propuestas que se encuentran en la literatura, y a la falta de una validación efectiva de sus alcances.

En este capítulo se presenta el Modelo para Desarrollar y Gestionar la Ingeniería de Requisitos MoDeMaRE, en el que se involucra conceptos de los métodos formales, la lógica, la matemática discreta y la planeación estratégica, integrados sistemáticamente para lograr una adecuada especificación. Se estructura en etapas que abarcan desde la comprensión del problema y el contexto, hasta la entrega del documento de la fase. Para cada una se describe las actividades necesarias para su cumplimiento, a la vez que el producto a generar. Todo el proceso es iterativo e incremental y se lleva a cabo involucrando procesos de Verificación y Validación, y del cumplimiento de una depende el progreso a la siguiente etapa.

1. DESCRIPCIÓN DEL MODELO

El producto final de la Ingeniería de Requisitos debe ser un documento que constituye la base para las siguientes fases del ciclo de vida del producto. Los requisitos especificados deben ser claros, coherentes, no-ambiguos, verificables, validables y modificables. MoDeMaRE (Figura 1) se compone de cinco etapas: 0) temprana, 1) de elicitación, 2) de desarrollo, 3) de gestión, y 4) de especificación. Además, se inscribe en la concepción de que la Ingeniería de Requisitos es parte integral del proceso de desarrollo de software, y que las pruebas del software deben ser una actividad paralela a todo el ciclo de vida del producto [7].

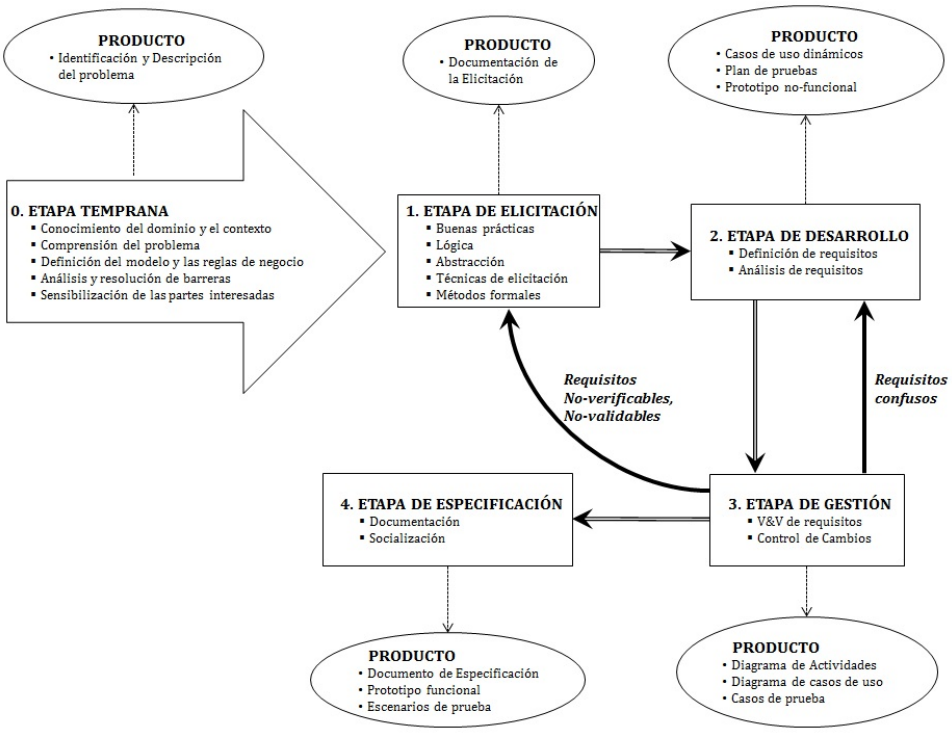

Figura 1. MoDeMaRE

1.1 Etapa Temprana

Los problemas relacionados con la Ingeniería de Requisitos se han reportado ampliamente en la literatura, especialmente los relacionados con la calidad del producto y con los costos y retrasos que generan una inadecuada gestión y administración [22]. En uno de esos estudios, Tveito y Hasvold [23] observan que existe una enorme brecha entre la realidad de las operaciones del día a día en la organización y el conocimiento del dominio que alcanzan de ella los desarrolladores. Los autores argumentan que esto se debe a la insuficiencia de los procesos de comprensión de la organización y a los malentendidos por falta de conocimiento del dominio.

De ahí la importancia de una Etapa Temprana, porque en ella se logra conocer el problema a la vez que analizar y comprender el dominio y el contexto de la organización, y el modelo de negocios. Este objetivo responde a las observaciones de los investigadores de que este proceso debe consistir en acciones estructuradas y repetibles, donde se manipule correctamente las actividades ingenieriles y de gestión [24]. Pero lamentablemente todavía no hay consenso en cuanto a los modelos adecuados para los procesos de Ingeniería de Requisitos que se puedan utilizar en diferentes contextos, porque cada uno tiene su propio sub-conjunto de estructuras. Esta propuesta es un acercamiento a ese consenso, porque sin importar el contexto lo que realmente vale la pena es comprender el problema, la organización y sus interrelaciones, y las partes interesadas. Esto se logra porque toda la Etapa 0 se desarrolla aplicando conceptos de Multidimensionalidad, Transdisciplinariedad y Complejidad para gestionar el conocimiento relacionado (Figura 2).

En éxito en la ejecución de toda actividad basada en la Gestión del Conocimiento requiere una comprensión del dominio en el que se produce. Este trabajo generalmente implica relaciones multidimensionales y transdisciplinares entre las diferentes y complejas fuentes del conocimiento, y para lograrlo se requiere una buena comprensión del dominio del problema y aplicar mucha creatividad. Debido a la diversidad de las partes interesadas involucradas en la Ingeniería de Requisitos, es común que inicialmente carezcan del conocimiento del dominio y del

contexto de la situación-problema, lo que le añade complejidad a los procesos. Como tal, el modelo para aplicar esta ingeniería debe partir de una fase estratégica, como la que aquí se propone, y de aplicarla adecuadamente requiere ingenio. Los ingenieros deben confiar en la base de conocimientos del dominio, a la vez que desarrollan otro con base en las visiones y análisis de los involucrados [25].

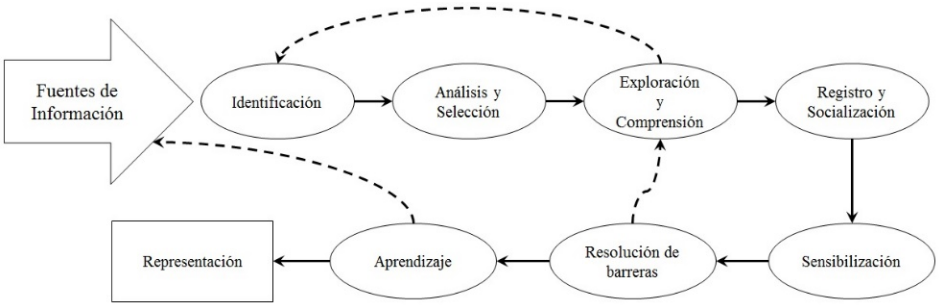

Figura 2. Etapa Temprana de MoDeMaRE

Los líderes de proyecto deberían intentar *utilizar* las partes interesadas vinculadas al problema como un medio para desarrollar el conocimiento del dominio mismo. A la vez que trabajar en estrecha colaboración con quienes se relacionan indirectamente en el contexto organizacional. Esto le ofrecería al equipo de trabajo el conocimiento de los modelos y reglas del negocio, para romper barreras y facilitar la comunicación entre todos [26]. Es necesario ser estratégico debido a que existe factores que pueden limitar la eficacia de esta estrategia en el desarrollo del conocimiento del dominio y el contexto.

En primer lugar, los usuarios o la organización pueden no querer compartir sus conocimientos o información con el equipo de ingenieros. Otro escenario podría ser que ese conocimiento del dominio es difícil de adquirir, debido a los tradicionales problemas de ambigüedad, las tecnologías de proveedores patentadas, u otras barreras de apropiación. Por eso es necesario diseñar estrategias de sensibilización y familiarización con la solución, antes de siquiera saber cuál es. En general, no debería ser difícil transferir el conocimiento del dominio y el contexto del problema cuando las estrategias están debidamente estructuradas e implementadas [27, 28].

En la medida que se descubre y construye conocimiento mediante la participación de las partes interesadas en esta Etapa Temprana, surge el aprendizaje práctico del dominio [29], pero requiere de mejoras y actualizaciones constantes, porque los involucrados no desarrollan confianza sino a medida que sienten que son importantes en el proceso. En esta medida van desarrollando mayor familiaridad con el equipo, el problema, la empresa y las tecnologías, y las barreas y prejuicios van disminuyendo. Alternativamente, los líderes del proceso pueden decidir desarrollar en el equipo capacidades de conocimiento del dominio y del contexto del problema. Esto se debe a que muchas veces los usuarios son nuevos, por lo que hay que invertir tiempo en capacitación que puede ser difícil, porque solamente el aprendizaje práctico en ese contexto les ofrece la realidad organizacional [29, 30). Pero si la empresa ha desarrollado capacitaciones en conocimiento del dominio se puede internalizar en el equipo de trabajo, lo que hace menos traumático el funcionamiento como tal.

Aquí hay que diferenciar entre la lógica de MoDeMaRE y otros puntos de vista frecuentes en la literatura acerca de este tema. En primer lugar, conocer el dominio y el contexto puede ser en sí mismo una forma para que los equipos adquieran conocimiento [26], y segundo, que este modo de proceso tiene una fuerte tendencia a facilitar la consecución del conocimiento, sin una verificación previa [31]. Aparte de la distinción que se hace en la presente propuesta entre el

conocimiento del dominio y otros tipos de conocimiento, la razón de ser de la Etapa Temprana es diferente de estas perspectivas, porque destaca y resuelve las dificultades de este aprendizaje, particularmente en la apropiación y credibilidad del mismo. Además, porque normalmente cada proyecto requiere nuevos conocimientos para comprender el problema, y ese escenario de aprendizaje se debe observar como una oportunidad. Por otra parte, se aprovecha los efectos de primer orden para desarrollar capacidades de conocimiento más fuertes [31], que serán la base para llevar a cabo las demás actividades.

Debido a que en esencia la meta de la Ingeniería de Requisitos es transformar las necesidades, potencialmente incompletas, inconsistentes y contradictorias, en un conjunto completo de requisitos de alta calidad, el líder debe tener en cuenta a todas las partes interesadas en el proceso [32]. Pero hay que reconocer que estos productos tienen dos esferas contextuales: 1) la del desarrollo de software a la medida, y 2) la del desarrollo para un mercado. En la primera situación el software se crea con un cliente específico en mente, y a menudo es posible tener contacto directo con él. Este contexto es diferente en la segunda situación, porque el producto se desarrolla para un conjunto de clientes que el equipo no conoce. Esto ha hecho que los proyectos software sean cada vez más complejos, y que los desarrolladores deban identificar los intereses de partes interesadas que provienen de una amplia gama de dimensiones y disciplinas [33].

La participación activa y permanente de todos los actores en las actividades de esta etapa proporciona en gran medida y determina el nivel de calidad del producto final. Por eso es importante encontrar y romper las barreras de comunicación y de participación, y trazar planes de sensibilización para que entiendan el rol que desempeñan en el proceso. Aunque algunos investigadores señalan que las incoherencias entre las partes y sus puntos de vista sobre los requisitos pueden ser deseables [34, 35], porque permiten una mejor elicitación, lo recomendable es tolerar algunas internas, pero comprendiendo la fuente y dándole importancia al impacto que puedan tener en la Ingeniería de Requisitos. Porque el éxito de los proyectos depende que desde esta etapa se siga procedimientos precisos y no-ambiguos. Por lo tanto, el objetivo es identificar tempranamente a los interesados y negociar y validar sus roles, antes de que se comprometa el producto de la fase, porque los factores emocionales son muy influyentes.

En este sentido poco se ha investigado acerca de la importancia de estos factores en la Ingeniería de Requisitos. Draper [36] hace un análisis desde la perspectiva de los requisitos para diseñar video juegos, tratando de identificar qué es lo que hace que las partes interesadas oculten o comuniquen sus emociones. Concluyó que una propiedad de la Ingeniería de Requisitos es lograr representar esas emociones como requisitos funcionales o no-funcionales, como una relación entre el software y los objetivos del usuario. Esto se define en MoDeMaRE como una clase importante del software, lo que es consistente con las conclusiones de este autor. Hassenzahl, Beu y Burmester [37] introducen las cualidades hedónicas, es decir, aquellas que no están relacionadas con la tarea en proceso, pero que están presentes por razones emocionales, y el repertorio de técnicas para medirlas.

Bentley et al. [38] investigan los factores afectivos en la Ingeniería de Requisitos, y concluyen que con los modelos actuales no es posible capturarlos de manera oficial, solamente como percepciones de las partes interesadas. En parte porque la técnica más utilizada es una motivación del mecanismo de encuestas. Señalan que no encontraron técnicas establecidas para la elicitación de los requisitos emocionales. Incluso Lawrence Chung [39], en su análisis detallado de los requisitos no-funcionales, no tiene en cuenta los problemas emocionales. Salen y Zimmerman [40], Laramee [41] y Marc Saltzzman [42] abordan las cuestiones emocionales y su comunicación en los equipos de requisitos. Si bien no abordan las prácticas directamente sobre una etapa

previa, las técnicas que describen y la retroalimentación que obtienen pueden incrementar la cobertura de estos factores en la elicitación. En un sentido más general, Donald Norman [43] describe numerosas prácticas para identificar los factores humanos, que podrían incorporarse fácilmente en la Ingeniería de Requisitos.

La adquisición del conocimiento del dominio y del contexto del problema y de las reglas del negocio, es una actividad importante en el contexto del análisis de dominio, porque es necesaria para organizar el conocimiento disponible sobre el contexto de aplicación que lo solucionará. El proceso que propone MoDeMaRE en la Etapa Temprana ayuda a identificar y organizar el conocimiento sobre un problema determinado, y a gestionarlo desde fuentes diversas, complejas, multidimensionales y transdisciplinares.

Este proceso comprende el aprendizaje con una preocupación subyacente acerca de su aplicabilidad, la recolección, la organización y el modelamiento del conocimiento dentro de un dominio específico. En el contexto de la Etapa Temprana la adquisición de conocimiento abarca aspectos similares a los propuestos en otros modelos para otros contextos [44], pero aquí es importante identificar, obtener y registrar la información que pertenece y proviene de la familia de sistemas relacionados, en lugar de visionarla como un sistema único.

Se podría pensar que desde la oferta de modelos y técnicas para identificar, obtener y representar conocimiento para productos software [45-57] esta es una situación bien resuelta, sin embargo, al considerar la cuestión de la adquisición de conocimiento, estos enfoques reconocen la importancia de la actividad, pero no proporcionan un proceso sistemático para la realización de la misma, como sí lo hace la Etapa Temprana de MoDeMaRE. En la propuesta se realizó un estudio a los enfoques actuales para el análisis del dominio y se estructuró una perspectiva simplificada para aplicarla en esta fase, como se muestra en la Figura 2. Primero se identifica las fuentes de conocimiento del dominio y el contexto del problema, la empresa y las partes interesadas, que se seleccionan en términos de relevancia, adecuación y coherencia. Bajo esta perspectiva priman dos características para su evaluación: 1) la elicitación del conocimiento, y 2) su representación.

1.2 Etapa de Elicitación

La Ingeniería de Requisitos es una actividad que tiene como objetivo descubrir, documentar y mantener el conjunto de necesidades del cliente. El uso del término ingeniería implica utilizar técnicas sistemáticas y repetibles para asegurar que esos requisitos sean completos, consistentes y relevantes [58]. Loucopoulos y Karakostas [59] subrayan la importancia del trabajo cooperativo en las actividades de esta fase, en particular el desarrollo de requisitos a través del análisis al problema, la documentación de las observaciones en una variedad de formas de representación y la comprobación de la exactitud del conocimiento adquirido [60, 61].

Zave [62] define a la Ingeniería de Requisitos como la rama de la Ingeniería del Software relacionada con los objetivos del mundo real para construir sistemas software. También se refiere a la relación de estos factores con las especificaciones precisas del comportamiento del producto y con su evolución a lo largo del tiempo, lo que motiva el desarrollo de un sistema, es decir, el por qué y el qué. Para MoDeMaRE también es importante hacer énfasis en la realidad de un mundo cambiante y en la necesidad de reutilizar las especificaciones parciales, como lo hacen los profesionales en otras disciplinas ingenieriles [63].

La elicitación es la actividad de capturar las necesidades de las partes interesadas y desde las diferentes dimensiones, que posteriormente se analizan para determinar las áreas que requieren

aclaración, relaciones lógicas o abstracción. Luego se documentan formalmente y por último se validan con todos los actores del proceso, para garantizar que las satisface el producto a desarrollar. Davis [64] sugiere que esta etapa es más que un ejercicio de descubrimiento que es absolutamente esencial, porque sin ella no sería posible construir adecuadamente el producto. Incluso, para los proyectos de desarrollo de software, en los que ningún cliente específico proporciona los requisitos, el desarrollador los debe elicitar implícitamente.

Esta fase se centra principalmente en examinar y reunir los requisitos y objetivos deseados para el sistema desde el punto de vista de los clientes, los usuarios, las limitaciones, el sistema operativo, el negocio, la comercialización, los estándares, las pruebas, la seguridad, entre otras dimensiones y disciplinas. La elicitación comienza con la identificación y recolección de los requisitos en bruto y de acuerdo con las diversas opiniones y recomendaciones. El objetivo es recoger diferentes puntos de vista acerca de las necesidades desde el negocio, el cliente, los usuarios, las limitaciones, la seguridad, la información, las normas, … y generalmente se lleva a cabo aplicando una técnica de elicitación, como la observación o la entrevista [65].

A menudo esos requisitos son mal comprendidos debido a que el equipo puede malinterpretar las necesidades, el domino y el contexto del problema, por lo que es importante aplicar adecuadamente la Etapa Temprana. Para lograrlo hay que tener en cuenta que las normas, los estándares y las restricciones juegan un papel importante en el desarrollo del sistema, y que la elicitación debe ser contextual. Para estructurar esta etapa en la propuesta MoDeMaRE, Serna y Suaza [21] llevaron a cabo una investigación en la que buscaron los modelos propuestos para realizar y documentar la elicitación de requisitos. Debido a que no encontraron uno que satisficiera los objetivos de su investigación, propusieron un nuevo modelo semi-formal [66], que se utiliza como base en la Etapa de Elicitación que describe MoDeMaRe. Estos autores optaron por seleccionar una serie de principios, considerados o no por los investigadores, como necesarios para un modelo de este tipo, y las mejores prácticas descritas en los trabajos que revisaron. Los componentes de ese modelo son:

1. *Mejores prácticas*. Tales como: 1) plantillas, utilizadas como medio para colectar información y documentar procesos. Algunos autores las utilizan como modelos de elicitación y negociación, y las aplican como complemento a las técnicas de elicitación, donde la documentación final se maneja directamente por la interacción de los participantes. 2) Esquemas, tales como diversos tipos de diagramas para observar factores de la elicitación de forma organizada. En general se utilizan para representar sub-sistemas, componentes de un sistema o las relaciones entre ellos. 3) Matrices de variables, utilizadas como una forma analítica de interpretar las interacciones de los factores en la capa inicial de la lógica de un programa.

   Para algunos autores las variables que interactúan y el comportamiento del sistema, solo son correctos cuando se documentan apropiadamente, pero en general se utilizan como complemento a las plantillas, porque también hacen uso de variables o campos para asociar y dividir la información. 4) Indicadores, para evaluar de qué forma la elicitación contribuye a mejorar la interpretación del problema, y como consecuencia deben introducir criterios que permitan mejorar el proceso. 5) Escenarios, utilizados para describir el comportamiento del sistema, para asegurar una mejor comprensión y colaboración entre las partes en la elicitación de requisitos, y para mantener información que puedan reconocer. 6) Diagramas, para mostrar la interacción de los requisitos con el sistema, lo cual añade valor para el usuario, porque especifican su comportamiento. Es decir, describen qué hace el sistema, no cómo lo hace. Son la representación simbólica o pictórica de un procedimiento administrativo.

2. *Principios*. Tales como: 1) lógica y abstracción, porque su utilidad radica en que permiten comprender, analizar y modelar tanto el problema como la posible solución, a la vez que les facilitan a los usuarios y clientes la comprensión de lo que están interpretando, por ejemplo, los requisitos. Ambos principios son necesarios que se aplicaron en la Etapa Temprana para comprender el contexto y el problema mismo, pero aquí son herramientas que facilitan encontrar la información necesaria para documentar la elicitación. 2) Métodos Formales, que son técnicas y herramientas basadas en principios y postulados matemáticos para especificar, diseñar, validar y verificar sistemas. Los principios que se tienen en cuenta en la estructuración de esta Etapa son: el cálculo proposicional [67]; las tablas de decisión [68]; la teoría de conjuntos; los lenguajes declarativos [69]; y el diseño por contrato [70].

Semi-formal se entiende como una agrupación de códigos de procedimiento que indican el tipo de descripción utilizada para documentar la información [71], que se centran en la creación de un modelo del sistema en una etapa determinada del ciclo de vida, y su objetivo es realizar transformaciones de modelos automáticos [72]. A diferencia de un modelo no-formal sus notaciones son intuitivas, permiten una mejor abstracción de los detalles y aplican metodologías estandarizadas y bien definidas [73]. Los modelos semi-formales son ampliamente utilizados en la industria del software, debido a que su semántica ayuda a evitar la ambigüedad, la inconsecuencia y la imprecisión, y a que se razonan formalmente [74, 74]. La Etapa de Elicitación de MoDeMaRE se estructura con el objetivo de diseñar el modelo de comportamiento del sistema. Aunque no se recomienda ni acoge una técnica de elicitación específica, se deja la decisión al líder del proyecto, porque cada problema es diferente y cada equipo tiene intencionalidades diferentes. Los pasos de esta Etapa son:

1. *Elaborar el diagrama de procesos*. Un diagrama de procesos se utiliza para mostrar las relaciones entre los principales componentes de un sistema, también para tabular valores de diseño de procesos para los componentes en diferentes modos de funcionamiento. Muestra la relación entre los sub-procesos del sistema e interpretado, a la vez que oculta los detalles de menor importancia, tales como responsabilidades y denominaciones de agentes. Como representación gráfica, en esta propuesta el diagrama de procesos se utiliza para visualizar el flujo de procesos, donde se puede observar las diferentes actividades del sistema y las conexiones entre ellas. Esto les permite a las partes interesadas una mejor comprensión del problema, y analizar la abstracción y el modelado de la solución.

2. *Aplicar cálculo proposicional*. Es un principio de los Métodos Formales que se utiliza para formalizar el lenguaje natural, con el que los usuarios expresan sus necesidades, y para estructurarlas en forma de proposiciones lógicas [76]. Las proposiciones se muestran en la lógica como objetos de un lenguaje formal mediante diferentes tipos de símbolos, que se concatenan de acuerdo con reglas recursivas para construir cadenas a las que se asigna valores de verdad. Al representar los requisitos de esta forma es posible verificar que las interacciones de los actores y el sistema respetan las reglas del negocio, y que en las actividades de prueba posteriores se obtendrá los valores de salida esperados. En la elicitación de requisitos el cálculo proposicional es importante para representar las necesidades como fórmulas matemáticas, y para facilitar la estructuración del plan de pruebas.

3. *Elaborar el diagrama de caminos*. Consiste en la representación del modelo ideal de comportamiento del sistema por medio de un grafo dirigido. Para este proceso se asigna un valor secuencial de nodo y una prioridad y dependencia de ejecución, de acuerdo con el diagrama de procesos, y en sus aristas se detallan las pre y post condiciones. Este grafo se convierte en una vista de la información de partida y facilita la construcción, lectura e

interpretación del documento de elicitación. Las proposiciones iteradas y representadas en el diagrama de caminos constituyen las acciones que los actores realizan sobre el sistema.

4. *Completar la plantilla de elicitación*. Las proposiciones iteradas y representadas en el diagrama de caminos son las acciones que un actor realiza sobre el sistema, y son un proceso que se lleva a cabo mediante una serie de interacciones conformadas igualmente por una serie de acciones. Además, esa relación genera una serie de acciones que el sistema ejecuta como respuesta a las solicitudes del actor. Este proceso de comunicación produce un cambio de estado en las variables, los valores, las bases de datos y el hardware, que también se debe describir. Para documentar este proceso de comunicación actor-sistema-actor, en esta Etapa se propone utilizar la plantilla de la Tabla 1 para capturar la información del proceso [21].

Tabla 1. Plantilla para documentar la elicitación de requisitos en MoDeMaRE [21]

Nombre Interacción				
Código Interacción				
Descripción Interacción				
Actor(es)				
Pre-condiciones				
Dependencias anteriores				
Camino Principal	Acción actor	Reglas del negocio	Acción/respuesta sistema	Cambios de estado
Caminos Alternativos				
Dependencias posteriores				
Post-condiciones				
Observaciones				

La plantilla contiene la información necesaria para generar el modelo de comportamiento del sistema: 1) *acción actor*, para detallar la acción que el actor realiza como parte de una interacción; 2) *reglas del negocio*, para verificar las operaciones, si los valores resultantes son los esperados y si los casos de prueba no violan las reglas establecidas; 3) *acción/respuesta sistema*, para describir las acciones o respuestas que el sistema ejecuta, ya sea como acciones internas o como respuesta a una acción previa del actor; 4) *cambios de estado*, para verificar que los cambios en la base de datos y las variables operacionales sean los esperados.

5. *Generar el reporte de requisitos*. Es un informe en el que se detalla el primer acercamiento a la descripción de requisitos, que se extrae desde las proposiciones, el tipo al que corresponde (funcional o no-funcional) y sus relaciones. Es una tabla que sirve de complemento a la plantilla y que juntas son el valor agregado para la siguiente etapa de MoDeMaRE y para el posterior documento de especificación. Debido a la volatilidad de los requisitos y al constante cambio en las necesidades de los clientes, todo el proceso de la Etapa de Elicitación se puede actualizar dinámicamente, porque sin importar que la acción sea *bottom-up* o *up-bottom* se reflejará fácilmente en los demás pasos.

1.3 Etapa de Desarrollo

El objetivo de esta Etapa es realizar un análisis detallado y retroalimentar el producto de la elicitación, para luego comparar los resultados con el carácter técnico del sistema y generar un

informe con los requisitos verificados y validados, necesarios para el desarrollo del software. La definición de requisitos es una parte importante de esta etapa, porque una definición incorrecta, inexacta o excesiva necesariamente dará lugar a retrasos en el plan de trabajo, desperdicio de recursos o insatisfacción del cliente. Por otro lado, el análisis de requisitos debe comenzar con los del negocio para traducirlos en requisitos del proyecto. Cuando no se aplica adecuadamente la Etapa Temprana, elicitar y desarrollar los requisitos del negocio puede ser injustificadamente costoso y tomar demasiado tiempo. Cuando se hace bien se puede llegar a los requisitos del proyecto de forma negociada en las conversaciones con los clientes o patrocinadores.

Las discusiones sobre métodos de definición o de análisis de requisitos necesarios deben ser específicos para el tipo de problema, el dominio y el contexto identificados en la Etapa Temprana, porque muchas empresas tienen técnicas específicas y probadas para hacerlo, pero pueden no brindar una definición completa y precisa de las necesidades para un proyecto específico. Mientras que los métodos pueden ser diferentes, los principios siguen siendo los mismos en todo tipo y tamaño de proyecto. Deben cubrir todo el alcance del mismo, ser amplios y exhaustivos y tener en cuenta las opiniones y necesidades de todos los interesados. Debido a que los requisitos confusos o no-verificables ni validables se pueden dejar por fuera del ámbito de aplicación de un análisis, es necesario que en esta etapa se defina con claridad la ambigüedad de los mismos. Por eso MoDeMaRE recomienda que la definición y el análisis de requisitos deben ser revisados y aprobados por todo el equipo involucrado antes de seguir con el trabajo.

Este proceso implica comunicación frecuente con los usuarios del sistema para determinar las expectativas y características específicas, la resolución de conflictos y la ambigüedad en los requisitos. En esto hay que evitar la fluencia de características, y documentar todos los aspectos del proceso de principio a fin. El objetivo es garantizar que el sistema final o el producto se ajustan a las necesidades del cliente, en lugar de intentar moldear las expectativas para que se amolden o adapten a los requisitos. Aquí vuelve a cobrar relevancia la Etapa Temprana, porque la definición y el análisis que se concibe en la Etapa de Desarrollo es un esfuerzo de equipo que requiere una combinación de hardware, software y factores humanos multidimensionales y transdisciplinares, por lo que se necesita buenas habilidades en el trato con las personas.

Mediante el proceso iterativo que se describe en la Figura 3 para la definición y el análisis, los requisitos se pueden refinar para determinar su impacto en los procesos del negocio, los sistemas relacionados y las modificaciones posteriores. Con esa información se puede aplicar esfuerzos futuros de diseño para satisfacer la evolución de la información, la integración de sistemas y para responder adecuadamente a las necesidades y retos del negocio.

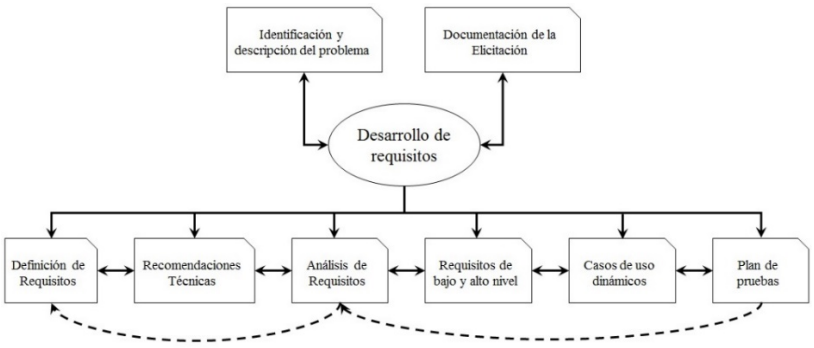

Figura 3. Etapa de Desarrollo de MoDeMaRE

Una cuestión importante que se debe tener en cuenta en esta Etapa es que los cambios en el software se producen por varias razones, por ejemplo, con el fin de fijar fallas, para agregar

nuevas características o para reestructurarlo y adaptarlo a cambios futuros [77]. Los cambios en los requisitos es una de las motivaciones más importantes para el cambio en el software. Ellos cambian desde el momento en que elicitan hasta que el sistema se vuelve obsoleto o se retira, lo que refleja que el sistema debe cambiar para mantenerse útil y seguir siendo competitivo en el contexto de la organización. Al mismo tiempo, estos cambios suponen un gran riesgo, porque pueden causar deterioro del software. Por lo tanto, los cambios en los requisitos deben ser capturados, gestionados y controlados cuidadosamente para asegurar la supervivencia del sistema desde el punto de vista técnico. Según Leffingwell y Widrig [78] los factores que pueden motivar cambios en los requisitos durante el desarrollo inicial como en la evolución son:

- Suponer que el sistema auto-resuelve los cambios sin importar la razón.
- Luego de que entienden mejor sus necesidades los usuarios cambian de opinión acerca de lo que quieren que haga el sistema. Esto puede suceder debido a que inicialmente no estaban seguros sobre lo que querían, o porque entran en escena nuevos usuarios.
- Cuando cambia el entorno del sistema, por ejemplo, el aumento de la velocidad y la capacidad de los computadores pueden afectar sus expectativas.
- Cuando se desarrolla un nuevo sistema y se libera, conduce a los usuarios a descubrir nuevos requisitos.

Este último factor es real y muy común, porque cuando un sistema se libera los usuarios se dan cuenta de que quieren características adicionales, que necesitan que los datos se presenten de otras formas, que hay nuevas necesidades para integrar el sistema con otros sistemas, y así sucesivamente. Todo esto genera nuevos requisitos. De acuerdo con *las leyes de la evolución del software* [79], un sistema debe adaptarse continuamente o hará cada vez menos satisfactorio su entorno. Los problemas surgen cuando las necesidades y los cambios en los requisitos no se manejan adecuadamente en el desarrollo [77]. Por ejemplo, el hecho de no hacer las preguntas correctas a las personas correctas en el momento adecuado en la Etapa de Desarrollo, conducirá muy probablemente a un gran número de cambios de requisitos en las fases posteriores del ciclo de vida. Además, si no se crea un proceso de prácticas de gestión de cambios puede significar que éstos no se puedan manejar oportunamente, o que si se implementan no tendrán un adecuado control. Esto es responsabilidad de la Etapa de Desarrollo, cuyas actividades más importantes son:

- *Definición de requisitos*. La gestión de requisitos es un enfoque sistemático para encontrar, documentar, organizar y hacerle seguimiento a los requisitos, y a los cambios que se puedan producir en todo el ciclo de vida del proyecto. Una buena gestión de requisitos ayuda efectivamente a garantizar que el producto final satisface las necesidades y expectativas de las partes interesadas. Una definición de requisitos efectiva significa simplemente averiguar qué hacer antes de empezar a hacerlo, teniendo en cuenta que el producto final debe adaptarse a las necesidades del cliente, en lugar de que éste se adapte al producto. A menudo los proyectos encuentran dificultades, porque carecen de requisitos definidos y documentados claramente. Por eso es que la Etapa de Elicitación es un paso clave para la gestión de requisitos y pertinente para todas las partes interesadas, para determinar y acordar los requisitos específicos agrupando procesos, técnicas de documentación, trazabilidad y expectativas de prueba.

  Para MoDeMaRe la definición de requisitos es una parte crucial del desarrollo del proyecto, porque una definición incorrecta, inexacta o excesiva de requisitos necesariamente dará lugar a retrasos en el programa, desperdicio de recursos, o la insatisfacción del cliente. Si los requisitos se definen mal luego de las Epatas 0 y 1, el proceso será injustificadamente costoso, o tomará demasiado tiempo, porque se tendrá que renegociar los requisitos. Muchas

industrias e investigadores proponen y aplican técnicas específicas para obtener una definición completa y precisa de los requisitos, y algunas escriben manuales para los usuarios como una forma de definirlos adecuadamente. Aunque estos métodos pueden ser diferentes, los principios siguen siendo los mismos en todo tipo y tamaño de proyecto. La definición de requisitos debe cubrir todo el alcance del proyecto, debe ser amplia y exhaustiva, y tener en cuenta las opiniones y necesidades de todos los interesados. Debido a que es fácil dejar detalles por fuera del ámbito de la definición u omitir claridades al respecto, dicha definición puede resultar ambigua. Por eso es que la definición de requisitos debe ser revisada y aprobada por el cliente antes de seguir con las demás actividades de la Etapa. Algunas buenas prácticas para realizar una definición de requisitos ajustada son:

- Examinar las necesidades y oportunidades del negocio.
- Escribir claramente los objetivos del proyecto.
- Conocer la diferencia entre deseos y necesidades.
- Negociar interactivamente con las partes interesadas la definición de requisitos.
- Llevar a cabo un análisis a fondo y exhaustivo de lo acordado.
- Documentar los resultados sin ambigüedades y con el detalle suficiente.
- Poner bajo control de versiones el documento logrado.

- *Análisis de requisitos*. El desarrollo y la recolección de requisitos es una actividad básica para cualquier organización que pretenda desarrollar productos software de calidad. Posteriormente, esos requisitos son rigurosamente analizados en el contexto del modelo del negocio. Además, tradicionalmente sucede que los requisitos en bruto identificados pueden ser contradictorios [80]. Por lo tanto, la negociación, los acuerdos, la comunicación y la priorización se convierten en actividades importantes de este análisis. Los requisitos analizados deben ser documentados, para permitir la comunicación con las partes interesadas y el futuro mantenimiento del sistema y de ellos mismos. Esta actividad también permite el refinamiento de la distribución del software y del diagrama de procesos, de los datos y de los dominios de comportamiento de que puede tratar el software. La priorización también es parte del análisis de requisitos del software.

- *Distribución y ubicación de requisitos*. El propósito de esta actividad es asegurar que todos los requisitos del sistema se cumplen completamente por un sub-sistema o por un conjunto de sub-sistemas, los cuales trabajan juntos para lograr los objetivos. Los requisitos de nivel superior se deben organizar jerárquicamente, de tal forma que ayuden a ver y gestionar la información en los diferentes niveles de abstracción. Los requisitos se descomponen hacia abajo hasta el nivel en el que puedan ser diseñados y probados. Por eso la asignación y la ubicación pueden realizarse mediante varios niveles jerárquicos, donde el nivel de detalle se incrementa a medida que el proceso desciende en la jerarquía. Es decir, los requisitos a nivel del sistema son de carácter general, mientras que en los niveles bajos de la jerarquía son muy específicos [81, 58]. Los requisitos del sistema del nivel superior se convierten en la base para la distribución y para la ubicación. Dado que se están desarrollando los requisitos de nivel de sistema, también se deben considerar los elementos a definir en la jerarquía [82].

  La *distribución* es una tarea arquitectónica para diseñar la estructura del sistema y para distribuir los requisitos del nivel superior a los sub-sistemas. Los modelos arquitectónicos proporcionan el contexto para definir la interacción entre las aplicaciones y los sub-sistemas, con el objetivo de satisfacer los requisitos. El objetivo del modelado arquitectónico es definir un marco robusto dentro del cual pueda desarrollarse las aplicaciones y los sub-sistemas componentes [83]. Cada requisito del nivel del sistema se asigna a uno o más elementos en el

siguiente nivel. La asignación también incluye la distribución de los requisitos no-funcionales para los elementos del sistema. Cada elemento necesitará un desglose de requisitos no-funcionales, por ejemplo, los requisitos de desempeño [4, 78, 82, 84]. Luego de que los requisitos funcionales y no-funcionales han sido asignados se puede crear un modelo que represente la interrelación entre los elementos del sistema, y establecer una base para el posterior análisis de requisitos y pasos del diseño.

La *ubicación* consiste en escribir los requisitos para los elementos de nivel inferior en respuesta a la distribución anterior. Cuando un requisito del sistema es distribuido a un sub-sistema, éste debe tener al menos uno de los requisitos que responda a esa asignación. Los requisitos de nivel inferior o muy cerca pueden parecerse a los de nivel superior, o ser muy diferentes cuando se reconoce una capacidad que el elemento de nivel inferior debe tener para cumplir con los requisitos de nivel superior. A menudo los requisitos de nivel inferior se conocen como requisitos derivados [82], y son los que deben ser impuestos en el (los) sub-sistema(s). Estos requisitos se derivan del proceso de descomposición de los sistemas y existen dos sub-clases: los del sub-sistema y los de la interfaz. Los primeros deben ser impuestos a los propios sub-sistemas, pero no necesariamente proporcionan un beneficio directo al usuario final; los segundos surgen cuando los sub-sistemas necesitan comunicarse entre sí para lograr un resultado global. Este resultado se necesita para compartir datos, capacidades o la utilidad de un algoritmo de cálculo [81].

En la Etapa de Desarrollo la identificación y la trazabilidad de requisitos debe garantizarse, tanto para los de nivel superior como entre los requisitos en el mismo nivel. El fundamento de las decisiones de diseño debe registrarse, con el fin de garantizar que haya suficiente información para las fases de verificación y validación, lo mismo que para la gestión de las modificaciones. En teoría se debe producir un sistema en el que todos los elementos estén completamente equilibrados u optimizados. Debido al calendario y a las limitaciones tecnológicas, en la realidad rara vez se logra un equilibrio completo [78, 85]. La distribución y ubicación comienzan como actividades multidisciplinares, es decir, pueden contener sub-sistemas de hardware, de software y de técnicas. Inicialmente se consideran como sub-sistemas, pero de diferentes disciplinas, y en iteraciones posteriores se consideran separadamente. Los productos que entrega la Etapa de Desarrollo son:

- *Casos de uso dinámicos*. Al final se obtiene una representación de los requisitos a través de casos de uso dinámicos, tal como se representa en el ejemplo de la Figura 4.

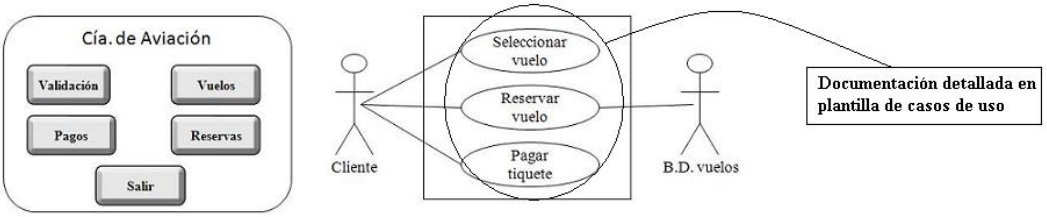

Figura 4. Casos de uso dinámicos en la Etapa de Desarrollo

- *Plan de pruebas*. Luego de que los requisitos están definidos, las diferentes partes involucradas deben ponerse de acuerdo acerca de su naturaleza. Esto significa que se deben asegurar de haber indicado los requisitos correctos –Validación– y que se han indicado correctamente –Verificación. Las actividades de Validación y Verificación incluyen validar los requisitos del sistema respecto a los requisitos en bruto, y verificar la correctitud de la documentación de los mismos. Las técnicas más comunes para validar requisitos son las revisiones con las partes

interesadas y la creación de prototipos. Los requisitos de software necesitan ser validados frente a los requisitos de nivel de sistema, y de esta forma se verifica la definición. La verificación incluye características tales como correctitud, consistencia, ambigüedades y comprensión de los requisitos. En la verificación y la validación el mecanismo de trazabilidad de los requisitos puede generar una pista de auditoría entre los requisitos de software y el código finalmente probado. La trazabilidad se debe mantener para los requisitos del nivel de sistema, entre los requisitos de software y para las fases posteriores. El resultado de la Etapa de Desarrollo de requisitos es un documento formal que incluye una línea base de lo convenido en los requisitos del proyecto [86].

- *Prototipo no-funcional*. Debido a que en este momento del proyecto es difícil realizar pruebas de usuario realmente útiles, porque la solución todavía no es lo suficientemente funcional para hacerlo, es necesario desarrollar un prototipo no-funcional con los avances y acuerdos para probar algo así como el análisis de la realización de tareas de la solución. Si bien satisfacer todas las partes interesadas o lograr una mezcla de datos pueden parecer temas obvios, no lo son. Mientras que en el proceso de codificación se creará una aplicación beta completamente funcional que se puede utilizar para las pruebas de usuario, el prototipo no-funcional se requiere para validar y verificar el proceso en cada Etapa de MoDeMaRe.

1.4 Etapa de Gestión

La falta de definición y gestión del cambio de requisitos es la causa más frecuente de fracasos en el desarrollo de software. Aunque la industria, los profesionales y los investigadores son conscientes de ello, gestionar adecuadamente los requisitos es difícil. La causa más común es el manejo inadecuado e ineficiente de los cambios, en parte porque le elicitación ocurre en ambientes informales y su documentación permanece estática durante el ciclo de vida de desarrollo, incluso cuando los requisitos cambian. Esto conduce a confusión, planificaciones perdidas, sobrecostos y usuarios y desarrolladores insatisfechos.

El objetivo de esta Etapa en MoDeMaRe es adoptar y gestionar el cambio, no evitarlo. La gestión *ad hoc* de requisitos es ineficaz, sobre todo en sistemas complejos, de misión crítica, de varios niveles, o distribuidos. Porque este tipo de proyectos a menudo son ejecutados por equipos distribuidos en virtud del tiempo y la presión del mercado, lo que complica aún más la gestión de requisitos. La solución propuesta es diseñar, planificar y ejecutar procesos desde el análisis de requisitos y la gestión del cambio, a través de la colaboración y gestión de pruebas. La planificación de las pruebas y la definición de casos de prueba tienen que basarse en los requisitos, a fin de garantizar que el software cumple plenamente las necesidades y expectativas de las partes interesadas. Las buenas prácticas para lograrlo son:

- Tener un repositorio de requisitos actualizado y coherente.
- Administrar los atributos de los requisitos mediante la definición de una persona responsable, el número de versión, el estado, la prioridad, el costo y el riesgo, entre muchas otras características.
- Hacerle seguimiento y comunicar los cambios a través de flujos de trabajo.
- Garantizar el acceso a los repositorios de todos miembros del equipo.
- Hacerle seguimiento al estado de los requisitos.
- Definir los casos de prueba de acuerdo con los requisitos.
- Tener un prototipo no-funcional para experimentar los casos de prueba.

El propósito de la gestión de requisitos es garantizar que la solución en desarrollo valida, verifica y responde a las necesidades y expectativas de las partes interesadas [87]. La gestión de requisitos incluye además la planificación de soporte e integración de los requisitos, así como las relaciones con otros datos relacionados en las etapas anteriores y los cambios de estos. Esta trazabilidad se utiliza en la gestión de requisitos para reportar los avances del proyecto en términos de cumplimiento, progreso, cobertura y consistencia. La trazabilidad también apoya la gestión del cambio, como parte de la gestión de requisitos en la comprensión de los impactos de los cambios, a través de los requisitos u otros elementos relacionados, y para facilitar la introducción de esos cambios [88].

Otra cuestión que implica la gestión de requisitos es la comunicación entre los miembros del equipo y las partes interesadas, y el ajuste de los requisitos a los cambios a lo largo del transcurso del proyecto [89]. Para evitar que una clase de requisitos anule otra, es fundamental una comunicación constante entre los miembros del equipo de desarrollo. Por ejemplo, en el desarrollo de software para aplicaciones internas, el negocio tiene ciertas necesidades fuertes que pueden hacer ignorar las necesidades del usuario, o creer que en la creación de casos de uso se están atendiendo.

La trazabilidad de requisitos se ocupa de documentar la vida de un requisito y el objetivo es poder rastrear el origen de cada requisito y de cada cambio realizado al mismo, por tanto, es necesario documentar todo el proceso a fin de lograr esta trazabilidad. Incluso se debe hacer seguimiento a las características implementadas después de haber sido desplegado y utilizado el requisito [90]. Debido a que los requisitos provienen de diferentes fuentes y a que éstas tienen diferentes expectativas para el producto, la trazabilidad a las características implementadas se debe remontar a la fuente desde donde se elicitó el requisito. Esto puede ser usado durante el proceso de desarrollo para darle prioridad o para determinar el valor del requisito para un usuario específico. También se puede utilizar después de la implementación, cuando los estudios de usuarios muestran que una característica, que no se utiliza, se debe atender prioritariamente.

En la Figura 5 se observa el proceso y las actividades que se proponen para la gestión de requisitos en MoDeMaRe.

Figura 5. Gestión de Requisitos

De esta manera será posible:

- Realizar trazabilidad completa con todos los otros componentes del proyecto, tales como los casos de prueba.
- Estructurar datos altamente personalizados al proyecto.
- Diseñar flujos de trabajo flexibles.
- Gestionar las tareas de forma integral.
- Mantener una base sólida de requisitos para auditorías y proyecciones.
- Estructurar requisitos de múltiples niveles.
- Atender las referencias multidimensionales.

- Integrar y gestionar documentación para facilitar la colaboración y el intercambio adicional.
- Construir e integrar diferentes puntos de vista de la comprensión del problema.
- Realizar las conexiones necesarias para gestionar la demanda.

En este proceso hay que tener claridad acerca de lo que es verificar y validar en el contexto de software, porque, aunque parezcan similares y muchos profesionales las confundan, tienen diferencias similares a las que existen entre la eficiencia y la eficacia en el contexto empresarial. En MoDeMaRe la validación es el proceso mediante el cual se confirma la integridad y la exactitud de los requisitos. Además, para garantizar que: 1) se logra los objetivos de negocio establecidos, 2) se satisface las necesidades de las partes interesadas, y 3) son claros y entendidos por los desarrolladores. La validación es esencial para identificar requisitos faltantes y para asegurar que reúnen ciertas características de calidad, y en la Etapa de Gestión se utiliza para asegurar que cada requisito es:

- Correcto: establece con precisión una necesidad.
- Claro: solamente tiene un significado posible.
- Factible: puede ser implementado dentro de las restricciones conocidas.
- Modificable: se puede cambiar fácilmente.
- Necesario: representa algo que realmente se necesita.
- Priorizable: clasificable de acuerdo con la importancia de inclusión.
- Trazable: se le puede hacer seguimiento a su vida útil.
- Verificable: su correcta implementación se puede determinar por medio de pruebas, análisis, inspecciones, o demostraciones.

Por otro lado, la verificación se refiere al proceso de garantizar la correcta aplicación de los requisitos, en otras palabras, la verificación responde a la cuestión de si se está haciendo lo correcto. En un ciclo de vida tradicional la verificación se utiliza para comprobar que los productos de cada fase cumplen los requisitos de la anterior. Este punto de vista es común en muchos textos de Ingeniería del Software, pero esto sería como asumir que los requisitos se pueden elicitar completamente desde el inicio de un proyecto, y que no cambiarán mientras se desarrolla el producto. Pero en la práctica los requisitos cambian a lo largo de todo el proyecto, en parte como reacción al proyecto mismo, porque el desarrollo de software hace posibles nuevas cosas y redefine continuamente las necesidades de los clientes. Por lo tanto, la validación y verificación son necesarias en todo el ciclo de vida.

En MoDeMaRe la verificación es un proceso mediante el cual se confirma que el producto que se está diseñando atiende y cumple totalmente los requisitos documentados. Consiste en realizar varias inspecciones, pruebas y análisis a lo largo de la Etapa de Gestión, para asegurar que el diseño, las iteraciones y el producto en construcción abordan plenamente los requisitos validados. Para evitar re-procesos los requisitos deben ser validados y verificados antes de la Etapa de Especificación. Si esto no es posible significa que inevitablemente deben ser elicitados nuevamente. Si se continúa sin tener en cuenta este detalle habrá una alta probabilidad de que no se descubran los requisitos faltantes o no-válidos, lo que provoca re-trabajo y que se incremente los costos.

Por eso es que en la Etapa de Gestión se propone validar y verificar cada requisito a medida que se diagrama, y de esta forma se mejora significativamente la calidad de cada uno. Cuando esta Etapa se aplica detenidamente, proporciona una base para la estimación de los costos y para la verificación del calendario del proyecto. Un requisito no-verificable es un requisito malo o

innecesario; si no se puede verificar, entonces no se puede diseñar o construir. Además, asegurar que los requisitos han sido validados y verificados es clave para reducir los riesgos del proyecto. Revisar cuidadosamente cada requisito asegura que es verificable, pero eliminarlo o elicitarlo nuevamente es una actividad importante en el proceso de la gestión de requisitos. Las actividades de la Etapa de Gestión garantizan que cada requisito es:

1. *Completo*. Un requisito se considera completo cuando todos sus componentes están presentes y cada uno está completamente desarrollado. Para determinar esta característica es necesario que: 1) no tenga indeterminaciones; 2) no haga referencia a funciones, entradas o salidas inexistentes; 3) no tenga elementos desaparecidos; 4) no le falten funciones; y 5) no tenga relaciones desaparecidas.

2. *Consistente*. Un requisito es consistente en la medida que sus componentes son compatibles entre sí o con los de los requisitos de su entorno. Esto se logra cuando: 1) presenta consistencia interna, 2) tiene consistencia externa, y 3) se le puede hacer trazabilidad.

3. *Factible*. Un requisito es factible cuando su especificación no supera los costos establecidos. Esto implica validar y verificar que es desarrollable y que se puede probar y mantener en el tiempo. Para esto debe: 1) satisfacer las necesidades de la ingeniería humana, 2) responder adecuadamente a la ingeniería de recursos, 3) atender las necesidades de la Ingeniería del Software, y 4) minimizar los riesgos asociados.

4. *Comprobable*. Un requisito es comprobable en la medida que se pueda identificar una técnica económicamente factible para determinar si se puede satisfacer. En este sentido deber ser: 1) específico, 2) no-ambiguo, y 3) cuantitativo.

1.5 Etapa de Especificación

Luego de progresar en las etapas anteriores se prepara un documento formal que contiene una descripción completa de los requisitos del sistema, hasta ese momento. El proceso de la especificación de requisitos es la actividad encargada de determinar qué funcionalidades del sistema se pueden desarrollar con los requisitos aprobados. En esta Etapa, y con la ayuda de la distribución, la ubicación y la derivación finales que son producto de las Etapas anteriores, se combina los requisitos no-funcionales con los funcionales. La especificación de requisitos consiste de una descripción completa de la finalidad y el contexto del producto en desarrollo. En ella se describe completamente lo que el software hará y cómo se espera que lo lleve a cabo.

El objetivo de la especificación es minimizar el tiempo y el esfuerzo requeridos por los desarrolladores para alcanzar las metas deseadas y el costo del desarrollo. Un buen documento define cómo interactuará la aplicación con la arquitectura de sistemas y con otros programas y usuarios en una amplia variedad de situaciones del mundo real. En ella se debe evaluar parámetros tales como velocidad de operación, tiempo de respuesta, disponibilidad, portabilidad, facilidad de mantenimiento, trazabilidad, seguridad y velocidad de recuperación de eventos adversos.

El documento de la especificación de requisitos se genera después de la identificación completa de los mismos, y describe el producto que se entregará antes de que inicie su desarrollo. En él se presenta una descripción completa del comportamiento del sistema a desarrollar. Incluye un conjunto de casos de uso que describe todas las interacciones que los usuarios tienen con el sistema/software [91]. Además de los casos de uso también contiene los requisitos no-

funcionales, que son los que imponen limitaciones en el diseño o la implementación. Los componentes más importantes de esta Etapa son:

1. *Prototipado de requisitos*. Es una técnica muy útil cuando el cliente no tiene claro las necesidades que el producto software debe satisfacer. Se trata de una primera versión de la apariencia que podría tener el producto, a la que se incorpora algunas características del sistema final, o que no se han estructurado completamente. Es un modelo o maqueta del sistema en desarrollo que tiene como objetivos comprender mejor el problema y las posibles soluciones, evaluar de mejor manera los requisitos y probar diferentes opciones de diseño. Los prototipos se caracterizan por tener poca funcionalidad y escasa fiabilidad, pero cumplen una importante tarea al permitirles a los ingenieros conocer la opinión y ayuda de los clientes, con lo que pueden mejorar el proceso de la Ingeniería de Requisitos y las fases posteriores del ciclo de vida del producto.

2. En la literatura el concepto *diagrama de caso de uso* no tiene una definición unificada. La mayoría de trabajos acogen la descripción de Jacobson [92], quien lo define como una serie de interacciones de un actor con el sistema. Aunque esta información es importante debido a que los casos de prueba escenifican las interacciones de los actores con el sistema, esa forma de aplicarlo se convierte en una caja negra, muchas veces no-comprensible en esta fase del ciclo de vida. En la Ingeniería de Requisitos participan usuarios, analistas y clientes por lo que se requiere mucha información para la elicitación de requisitos, que ese diagrama no proporciona ni está contenida en la plantilla de documentación, convirtiéndose en un modelo estático del sistema que no representa su comportamiento ideal.

   En MoDeMaRe se asume que un diagrama de casos de uso representa una interfaz sobre la que el actor interactúa con el sistema; además, que es un proceso que realiza en él mediante una serie de interacciones, conformadas igualmente por una serie de acciones. Esa relación actor-sistema debe generar una serie de acciones-respuesta del sistema a las solicitudes de las acciones del actor. Este proceso de comunicación genera cambios de estado en variables, valores, bases de datos y hardware, que también se deben documentar en la especificación.

3. *Escenarios de prueba*. Los escenarios se diseñan para representar tanto las situaciones típicas como las inusuales que pueden ocurrir en la aplicación [93]. De acuerdo con Kaner [94] un escenario es una historia hipotética, utilizada para ayudar a una persona a comprender un problema o sistema complejo. Para Ryser y Glinz [95] un *escenario* es un conjunto ordenado de interacciones entre socios, generalmente entre un sistema y un conjunto de actores externos a él. Puede comprender una secuencia concreta de pasos de interacción –instancias– o un conjunto de pasos posibles de interacción –escenario tipo. Un escenario de prueba ideal debe tener las siguientes características:

   - Es una historia acerca de cómo utilizar el programa incluyendo información de las motivaciones de las partes involucradas.
   - La historia debe ser motivadora y los involucrados pueden influenciar para desarrollar pruebas bajo un escenario determinado, ya que están motivados a hacerlo debido a la posibilidad de encontrar errores en dicho escenario.
   - La historia debe ser creíble, podría ocurrir no solamente en el mundo real y los involucrados deben creer que algo así probablemente puede suceder.
   - La historia implica un uso complejo del programa, un entorno complejo o un conjunto complejo de datos.

- Los resultados deben ser fáciles de evaluar. Esto es valioso para todas las pruebas, pero especialmente importante para los escenarios, debido a que en sí son complejos.

Un buen conjunto de escenarios debe dejar al probador preguntándose qué sería lo más verosímil o probable, y lo obliga a pensar más y a ampliar el punto central del escenario –para aprender más acerca de futuros alternativos–, con lo que tomará mejores decisiones. Por supuesto, los escenarios de este tipo son difíciles de construir y los estudios actuales no se comprometen ampliamente con ellos. Los escenarios deben educar para probar los requisitos, no para tratar de encontrar una solución determinada. Un escenario mal diseñado hace que al probador le sea difícil decidir cuál será el más probable o cuál logrará el mejor resultado. El mejor uso de los escenarios de prueba en esta Etapa es para lograr una comprensión cercana de las necesidades del sistema, y no para tratar de predecir el futuro funcionamiento del producto. Uno de los formatos más utilizados para realizar este documento es el estándar 830-1998 de IEEE [96].

2. CONCLUSIONES

La Ingeniería de Requisitos es la fase con la que comienza el ciclo de vida del proceso de la Ingeniería del Software. En ella se elicitan, comprenden y especifican las necesidades de las partes interesadas para desarrollar productos software de calidad, por lo que la ejecución de los procesos de esta fase merece mayor atención en las prácticas industriales. En este trabajo se propone un modelo semi-formal para gestionar la Ingeniería de Requisitos, que se puede utilizar para desarrollar procesos de software con el objetivo de producir mejores productos. MoDeMaRE es el resultado de un proceso de investigación, aplicación, verificación y comparación de las diferentes propuestas en la literatura para gestionar esta fase del ciclo de vida del software. Como resultado de ese proceso se originaron varios productos [20, 21] en los que se describe los pasos y resultados previos a la estructuración y formulación del modelo.

El modelo propuesto se puede utilizar en grandes procesos de desarrollo de software, donde los requisitos cambian continuamente, y presenta una nueva visión acerca de la gestión y planificación de requisitos en la que se tiene en cuenta esa naturaleza cambiante. Las actividades de ingeniería de requisitos en el modelo, como la elicitación, la documentación y la verificación y validación, están estrechamente interconectadas a las fases de desarrollo de software; además, este modelo puede ayudar a recuperar y a modificar los requisitos siempre que cambien en cualquier fase del desarrollo de software. El desarrollador de software puede fácilmente suplir los requisitos requeridos y modificados, aplicando la actividad de planificación, gestión y administración. Usando este modelo, el desarrollador puede diseñar productos con la capacidad de evitar y/o gestionar los nuevos requisitos. Este modelo es iterativo por naturaleza, lo que permite la creación de prototipos de los requisitos con mayor participación de los usuarios. Usando este modelo la organización puede permitirles a los usuarios cambiar los requisitos durante el proceso de desarrollo de software. En consecuencia, el documento se puede modificar y los requisitos modificados se pueden re-implementar en el proceso de desarrollo del producto.

El modelo que aquí se propone reúne las buenas prácticas encontradas en investigaciones previas, y potencializadas con aportes desde otras áreas del conocimiento, tales como los métodos formales, la matemática discreta, la lógica, la abstracción, y otras relacionadas. Esto se debe a que los problemas actuales son más complejos y complicados que antes, la seguridad se ha convertido en un asunto clave, las modificaciones son más cotidianas, y los tiempos de entrega de los productos software se han acortado drásticamente. Además, el modelo se estructura para hacerles frente a las debilidades encontradas en otros propuestos hasta el momento, porque todavía no brindan un soporte inteligente para la Ingeniería de Requisitos, y aún no cubren todas

las cuestiones en el desarrollo y la gestión de mismos, sin embargo, los resultados iniciales son prometedores. En términos generales se puede decir que:

- Un modelo para gestionar la Ingeniería de requisitos debe definir actividades que tengan como objetivo identificar requisitos, proporcionando actividades de soporte que contribuyan al éxito y a la calidad de los mismos.

- Para los ingenieros la flexibilidad en la implementación y la integración de los métodos existentes influye en el éxito del enfoque que adoptan. La ausencia de asociación para notaciones y métodos particulares es un factor clave en la aceptación del enfoque.

- Al principio la introducción de nuevos conceptos puede causar problemas, pero una sesión introductoria, para que los usuarios se familiaricen con los conceptos, puede contribuir al éxito de la implementación del modelo.

- Se necesita un enfoque iterativo en Ingeniería de Requisitos para que estos evolucionen. Sin embargo, se requiere una guía para determinar el comienzo o fin de cada etapa de esta fase.

- Los ingenieros de requisitos deben buscarlos activamente, en lugar de enfatizar en la elicitación desde fuentes conocidas. La elicitación puede ocurrir cuando sea posible identificar la fuente.

REFERENCIAS

[1] Hull E. et al. (2011). Requirements Engineering. Springer.
[2] Ropohl G. (1999). Philosophy of socio-technical systems. Society for philosophy and technology 4(3), 59-71.
[3] Stevens R. et al. (1998). Systems Engineering - Coping with complexity. Prentice Hall.
[4] Kotonya G. y Sommerville I. (1998). Requirements Engineering: Process and techniques. John Wiley.
[5] Zave P. (1997). Classification of research efforts in Requirements Engineering. ACM Computing Surveys 29(4), 315-321.
[6] Jureta I. et al. (2006). A more expressive soft goal conceptualization for quality requirements analysis. En D. Embley et al. (Eds.), ER 2006 (281-295). Springer.
[7] Serna E. (2012). Prueba del software: Más que una fase en el ciclo de vida. Revista de Ingeniería 35, 34-40.
[8] Andriole S. (1996). Managing Systems Requirements: Methods, tools and cases. McGraw-Hill.
[9] Juristo N. et al. (2002). Is the European industry moving toward solving Requirements Engineering problems? IEEE Software 9(6), 70-77.
[10] Martin S. et al. (2002). Requirements engineering process models in practice. En 7th Australian workshop on requirements engineering. Melbourne, Australia.
[11] Loucopoulos P. y Karakostas V. (1995). System Requirements Engineering. McGraw-Hill.
[12] Nguyen L. y Swatman P. (2003). Managing the requirements engineering process. Requirements engineering 8(1), 55-68.
[13] Houdek F. y Pohl K. (2000). Analyzing Requirements Engineering processes: A case study. En 11th international workshop on database and expert systems applications. Greenwich, UK.
[14] Hofmann H. y Lehner F. (2001). Requirements Engineering as a success factor in software projects. IEEE Software 18(4), 58-66.
[15] Lowe, D. y Eklund, J. (2001). Development issues in specification of web systems. Proceedings of 6th Australian workshop on requirements engineering (4-13). Sydney, Australia.
[16] Emam K. y Madhavji N. (1995). A field study of requirements engineering practices in information systems development. En 2nd international symposium on requirements engineering. York, UK.
[17] Chatzoglou P. (1997). Factors affecting completion of the requirements capture stage of projects with different characteristics. Information and Software Technology 39(9), 627-640.

[18] Siddiqi J. (1996). Requirement Engineering: The Emerging Wisdom. IEEE Software 13(2), 15-19.
[19] Pandey D. et al. (2008). Impact of Requirement Engineering Practices in Software Development Processes for Designing Quality Software Products. En National Conference on NCAFIS, DAVV. Indore, India.
[20] Serna E. y Serna A. (2016). Development and management requirements: Results of a literature review. En 8th Euro American Conference on Telematics and Information Systems. Cartagena, Colombia.
[21] Serna M. y Suaza J. (2016). Document requirements elicitation: A systematic review. Ingeniare - Revista Chilena de Ingeniería 24(4), 703-714.
[22] Hall T. et al. (2002). Requirements problems in twelve companies: An empirical analysis. IEEE proceedings software 149(5), 153-160
[23] Tveito A. y Hasvold P. (2002). Requirements in the medical domain: Experiences and prescriptions. IEEE Software (Nov-Dec), 66-69.
[24] van Lamsweerde A. (2000). Requirements Engineering in the year 00: A research perspective. En 22nd International conference on software engineering. Limerick, Ireland.
[25] Parmigiani A. (2007). Why do firms both make and buy? An investigation of concurrent sourcing. Strategic Management Journal 28, 285-311.
[26] Rothaermel F. et al. (2006). Balancing vertical integration and strategic outsourcing: Effects on product portfolios, new product success, and firm performance. Strategic Management Journal 27(11), 1033-1056.
[27] Darr E. et al. (1995). The Acquisition, transfer, and depreciation of knowledge in service organizations: productivity in franchises. Management Science 41(11), 1750-1762.
[28] Argote L. y Ingram P. (2000). Knowledge transfer in organizations: A basis for competitive advantage in firms. Organizational Behavior and Human Decision Processes 82, 150-169.
[29] Arrow K. (1962). The economic implications of learning by doing. Review of Econ. Studies 29, 155-173.
[30] Leiblein M. y Miller D. (2003). An empirical examination of transaction- and firm-level influences on the vertical boundaries of the firm. Strategic Management Journal 24(9), 839-859.
[31] Mayer K. y Salomon R. 2006. Capabilities, Contractual Hazard and Governance: Integrating Resource-Based and Transaction Cost Perspectives. Academy of Management Journal 49, 942-959.
[32] Pouloudi A. y Whitley E. (1997). Stakeholder identification in inter-organizational systems: Gaining insights for drug use management systems. European journal of information systems 6, 1-14.
[33] The Royal Academy of Engineering. (2004). The challenges of complex IT projects. The report of a working group from the Royal academy of engineering and the British computer society. BCS.
[34] Hunter A. y Nuseibeh B. (1997). Analyzing inconsistent specifications. En 3rd international symposium on requirements engineering. Annapolis, USA.
[35] Menzies T. et al. (1999). An empirical investigation of multiple viewpoint reasoning in requirements engineering. En IEEE international symposium on requirements engineering. Limerick, Ireland.
[36] Draper S. (1999). Analysing fun as a candidate software requirement. Personal Technology 3(1), 1-6.
[37] Hassenzahl M. et al. (2001). Engineering joy. IEEE Software 18(1), 70-76.
[38] Bentley T. et al. (2002). Putting some emotion into requirements engineering. En 7th Australian Workshop on Requirements Engineering. Melbourne, Australia.
[39] Chung L. (2000). Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers.
[40] Salen K. y Zimmerman E. (2004). Rules of Play: Game Design Fundamentals. MIT Press.
[41] Laramee F. (2002). Game Design Perspectives. Charles River Media, Inc.
[42] Saltzzman M. (2000). Game Design Secrets of the Sages. Macmillan Publishing.
[43] Norman D. (1988). The Design of Everyday Things. Doubleday Books.
[44] Arango G. y Prieto R. (1991). Domain Analysis Concepts and Research Directions. ACM SIGSOFT Software Engineering Notes 20(SI), 206-214.
[45] Prieto R. (1987). Domain Analysis for Reusability. En Eleventh Annual International Computer Software and Applications Conference. Tokyo, Japan.
[46] Prieto R. (1993). Software Reusability, Classification and Domain Analysis. En VII Simpósio Brasileiro de Educação em Solos. Rio de Janeiro, Brazil.
[47] Prieto, R. (1994). Software Reuse: From Concepts to Implementation. En Third International Conference on Software Reuse. Rio de Janeiro, Brazil.

[48] Cohen S. (1991). Process and Products for Software Reuse and Domain Analysis. En Fourth Annual Workshop on Software Reuse. Syracuse, USA.
[49] Neighbors J. (1992). The Commercial Application of Domain Analysis. En Fifth Annual Workshop on Software Reuse. Palo Alto, California.
[50] Leite J. (1994). Draco-Puc: A Technology Assembly for Domain Oriented Software Development. En Third International Conference on Software Reuse. Rio de Janeiro, Brazil.
[51] Gomaa H. (1994). A Prototype Domain Modeling Environment for Reusable Software Architectures. En Third International Conference on Software Reuse. Rio de Janeiro, Brazil.
[52] Gomaa H. (1995). Domain Modeling Methods and Environments. En Symposium on Software Reusability. Seattle, Washington.
[53] Arango G. et al. (1994). Software Reusability. Ellis Horword.
[54] Simos M. (1994). An Introduction to Organization Domain Modeling. En Third International Conference on Software Reuse. Rio de Janeiro, Brazil.
[55] Electronic Systems Center. (1996). Organization Domain Modeling Guidebook. Technical Report: STARS-VC-A025/001/00. Manassas, USA.
[56] Jacobson I. et al. (1997). Software Reuse – Architecture, Process and Organization for Business Success. ACM Press.
[57] Griss M. et al. (1998). Integrating Feature Modeling with the RSEB. En Fifth International Conference on Software Reuse. Victoria, Canada.
[58] Sommerville I. y Sawyer P. (1997). Requirements Engineering - A Good Practice guide. John Wiley.
[59] Loucopoulos P. y Karakostas V. (1995). System Requirements Engineering. McGraw-Hill.
[60] Pohl K. (1994). The Three Dimensions of Requirements Engineering: A Framework and its Applications, Information systems 19(3), 243-258.
[61] Pohl K. (1996). Process Centred Requirements Engineering. John Wiley.
[62] Zave P. (1997). Classification of Research Efforts in Requirements Engineering. ACM Computing Surveys 29(4), 315-321.
[63] Nuseibeh B. y Easterbrook S. (2000). Requirements Engineering: a Roadmap. In A. Finkelstein (Ed.), The future of software engineering (pp. 120-143). ACM Press.
[64] Davis A. (1993). Software Requirements: Objects, Functions, and States. Prentice Hall.
[65] Serna E. (2012). Analysis and selection to requirements elicitation techniques. En Séptimo Congreso Colombiano de Computación. Medellín, Antioquia.
[66] Serna E. y Suaza J. (2020). REDOC: Model for Documenting Requirements Elicitation. Computer Science - Research and Development. In press.
[67] Soares M. y Sousa D. (2012). Analysis of Techniques for Documenting User Requirements. Lecture Notes in Computer Science 7336, 16-28
[68] Smith C. y Williams L. (2003). Best Practices for Software Performance Engineering. En 29th International conference of Computer Measurement Group. Dallas, USA.
[69] Lloyd J. (1994). Practical advantages of declarative programming. En Joint Conference on Declarative Programming. Valencia, Spain.
[70] Mitchell R. y McKim J. (2001). Design by Contract by Example. Addison Wesley.
[71] Matta A. et al. (2004). Semi-Formal and Formal Models Applied to Flexible Manufacturing Systems. Lecture Notes in Computer Science 3280, 718-728.
[72] Ehrig H. et al. (2000). Semi-Formal and Formal Specification Techniques for Software Systems. Technical University Berlin.
[73] Snook C. y Butler M. (2006). UML-B: Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology 15(1), 92-122.
[74] Harel D. y Rumpe B. (2004). Meaningful modeling: what's the semantics of "semantics"? Computer 37(1010), 64-72.
[75] Locarti I. y Unguillo F. (2014). Analysis to design test cases. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 4(1), 13-20
[76] Grassmann W. y Tremblay J. (1998). Matemática discreta y lógica. Pearson Education.
[77] Mockus A. y Votta L. (2000). Identifying reasons for software changes using historic databases. En International Conference on Software Maintenance. San Jose, USA.
[78] Leffingwell D. y Widrig D. (1999). Managing software requirements - A unified approach. Addison.

[79] Lehman M. et al. (1997). Metrics and laws of software evolution - The nineties view. En 4th International Software Metrics Symposium. Albuquerque, USA.
[80] Yu Y. et al. (2008). From goals to high variability software design. En A. An et al. (Eds.), Foundations of Intelligent Systems (pp. 1-16). Springer.
[81] Parviainen P. et al. (2003). Requirements Engineering - Inventory of Technologies. VTT Publications.
[82] Royce W. (1987). Managing the Development of Large Software Systems. En 9th International Conference Software Engineering. California, USA.
[83] da Silva L. y Leite J. (2006). Generating Requirements Views: A Transformation-Driven Approach. En 3rd Workshop on Software Evolution through Transformations: Embracing the Change. Natal, Brazil.
[84] Kotonya G. y Sommerville I. (1996). Requirements Engineering with Viewpoints. Software Engineering Journal 11(1), 5-11.
[85] van Lamsweerde A. (2001). Goal-Oriented Requirements Engineering: A Guided Tour. En 5th IEEE international Symposium on Requirements Engineering. Toronto, Canada.
[86] Gilb T. (2003). Competitive Engineering. Addison-Wesley.
[87] Stellman A. y Greene J. (2005). Applied software project management. O'Reilly Media.
[88] Jallow A. et al. (2008). Lifecycle approach to requirements information management in construction projects: State-of-the-art and future trends. En 24th Annual Conference of Association of Researchers in Construction Management. Cardiff, Wales.
[89] PMI. (2013). A Guide to the Project Management Body of Knowledge. Project Management Institute.
[90] Gotel O. y Finkelstein A. (1994). An analysis of the requirements traceability problem. En First International Conference on Requirements Engineering. Colorado Springs, USA.
[91] Leite J. y Freeman P. (1991). Requirements Validation Through Viewpoint Resolution. Transactions of Software Engineering 12(2), 1253-1269.
[92] Jacobson I. (1992). Object Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley.
[93] Guru 99. (2010). What is Test Scenario? Template with Examples. Recuperado: https://www.guru99.com/test-scenario.html
[94] Kaner, C. (2003). An introduction to scenario testing. Florida Tech.
[95] Ryser J. y Glinz M. (1999). A practical approach to validating and testing software systems using scenarios. Third International Software Quality Week Europe. Brussels, Belgium.
[96] IEEE. (1998). 830-1998- IEEE Recommended Practice for Software Requirements Specifications. IEEE-SA Standards Board.

CAPÍTULO XVII
Documentar la elicitación de requisitos: Una revisión sistemática[1]

Edgar Serna M.[1]
Jorge Hernán Suaza J.[2]
[1]Instituto Antioqueño de Investigación
[2]Instituto Tecnológico Metropolitano

Diferentes investigadores han propuesto técnicas y modelos para elicitar requisitos, y la mayoría describe procesos y recomendaciones para capturarlos, pero muy pocos detallan cómo documentarlos en esta etapa. Aunque en la comunidad se reconoce ampliamente la importancia de la Ingeniería de Requisitos y de la Elicitación como una etapa importante de esta fase de la Ingeniería del Software, se ha presentado pocos trabajos que oriente a los ingenieros a documentar esa elicitación. La importancia de una adecuada documentación de esta etapa radica en que permite una mayor comprensión de las necesidades del cliente y el usuario, y les ayuda a los ingenieros a percibir de mejor forma el problema y a modelar una solución que se refleje adecuadamente en la Especificación. En este trabajo se presenta los resultados de una revisión sistemática de la literatura, orientada a conocer, analizar y comparar las propuestas para documentar la Elicitación de Requisitos. Se consultaron 73 trabajos en las bases de datos, de los cuales 18 conforman la muestra final. La conclusión es que falta más investigación, porque ninguno de los trabajos analizados describe una propuesta directa y completa para hacerlo.

[1] Publicado en Ingeniare. Revista chilena de ingeniería 24(4), 703-714. 2016.

INTRODUCCIÓN

Debido al tipo de problemas que atienden los ingenieros de software el proceso para solucionarlos es complicado, y requiere dedicación y tiempo. El procedimiento generalmente aceptado es el ciclo de vida de la Ingeniería del Software, que consiste en aplicar principios científicos de ingeniería en cada una de las fases, para generar una solución elaborada de software y para proyectar su subsecuente mantenimiento [1]. Este ciclo está conformado por las fases de: Ingeniería de Requisitos, Diseño, Desarrollo, Implementación y Mantenimiento, pero es conveniente tener en cuenta que las Pruebas y la Gestión de la Calidad deben ser paralelas a todo el proceso [2].

Por su parte, la Ingeniería de Requisitos se estructura en etapas cuyo objetivo es comprender, estructurar y documentar las necesidades que los usuarios desean satisfacer con el producto, y que posteriormente se traducirán en un conjunto de sentencias precisas, no ambiguas, que se utilizarán para desarrollar el sistema [3-4], y que se conocen como requisitos. El profesor Serna propone el Modelo para Desarrollar y Gestionar la Ingeniería de Requisitos MoDeMaRE (que hace parte de este libro), con el objetivo de aportar al conocimiento para desarrollar y gestionar la Ingeniería de Requisitos, y que está conformado por cinco etapas: 0. Temprana, 1. Elicitación, 2. Desarrollo, 3. Gestión, y 4. Especificación, como se observa en la Figura 1.

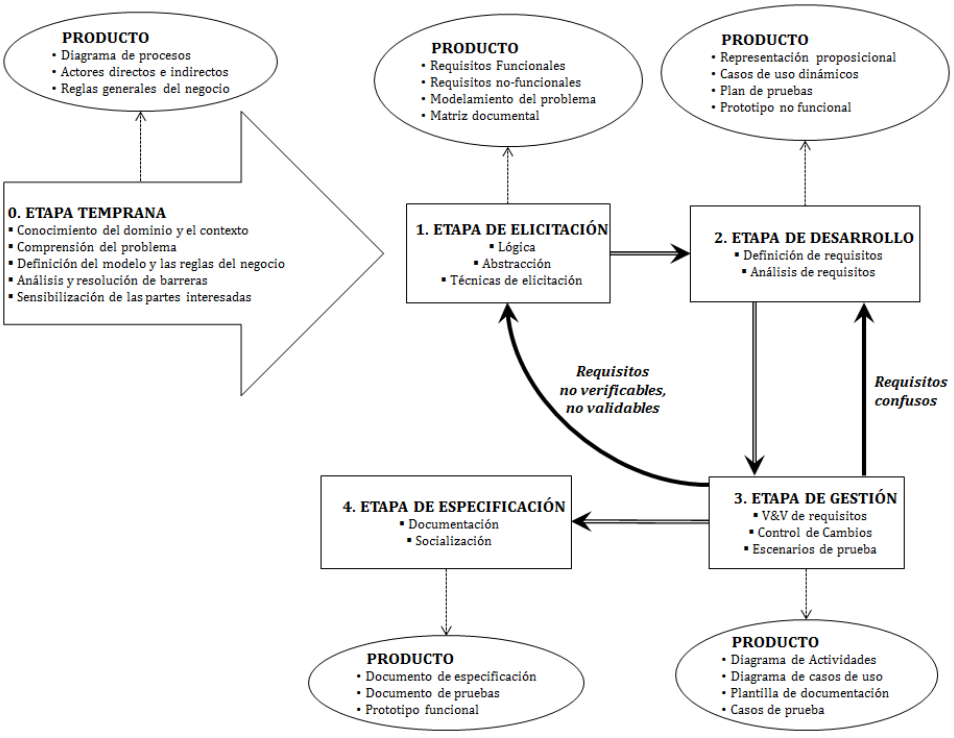

Figura 1. MoDeMaRE

De acuerdo con este investigador, en la etapa Temprana se reconoce y comprende el contexto del problema; en la Elicitación se identifica, modela y documenta las necesidades del cliente; en el Desarrollo se analiza y define esas necesidades; en la Gestión se validan y verifican; y en la Especificación se estructuran en un documento formal que constituye la base de la siguiente fase del ciclo de vida. En la elicitación se aplica diferentes técnicas para identificar las necesidades generalmente descritas en lenguaje natural, y por tanto con ambigüedades, lo que dificulta el proceso para las siguientes etapas. Las actividades se orientan a identificar las discrepancias entre las partes interesadas, con el propósito de definir claramente los requisitos [5]; pero esa identificación se convierte en un problema, porque no siempre están disponibles ni son claros

para el analista, además, porque la fuente es el conocimiento de seres humanos, que no es fácil de representar o traducir a un lenguaje de comprensión generalizado.

Para identificar y modelar el problema que se va a resolver el analista debe comprenderlo en contexto, y ubicarlo en un dominio. Para lograrlo utiliza desde la primera etapa las reglas generales del negocio y de los actores involucrados, directa o indirectamente. Pero el alcance del sistema, la falta de comprensión del problema por las partes interesadas, y la volatilidad de los requisitos complican su interpretación y modelamiento. Es por esto que las diferentes etapas de la Ingeniería de Requisitos son complicadas y exigentes, e implican procesos sistemáticos e iterativos en los que el analista debe recurrir a la lógica, la abstracción, y a sus capacidades de modelamiento, con el objetivo de hacer una representación mental del problema e iniciar el proceso de solución. Por lo tanto, en la etapa temprana es importante saber observar, saber preguntar, saber escuchar y saber representar de diferentes formas, y aplicar modelos lógicos y abstractos, porque de esta manera es posible eliminar la ambigüedad y las dificultades que se presentan cuando el cliente describe sus necesidades, lo que requiere un proceso juicioso y estructurado de documentación.

Documentar adecuadamente la elicitación brinda un mejor nivel de seguridad y comprensión del problema para abordar las demás fases del ciclo de vida, porque, debido a la volatilidad de los requisitos, se puede consultar cada vez que no se comprenda alguno en la especificación. Esto es necesario para que los proyectos software no excedan los tiempos ni los costos inicialmente establecidos, y para satisfacer de mejor forma la fiabilidad y seguridad esperada del producto. Pero, de acuerdo con la revisión a los modelos y metodologías propuestos para elicitar requisitos, estos principios no se tienen en cuenta, y la mayoría de autores pretende pasar a la especificación directamente, sin tener un documento adecuado de la elicitación. Esto se evidencia en los diferentes procesos de re-ingeniería que realizan los ingenieros en el Diseño del producto, luego de que se ha ejecutado la Ingeniería de Requisitos [6].

Con el objetivo de conocer y analizar las propuestas de la comunidad para documentar la Elicitación de Requisitos, se hizo una revisión sistemática de la literatura para analizarlas y compararlas. Este capítulo muestra los resultados en los que se describe los enfoques y procedimientos relacionados. En el procedimiento se consultó diferentes bases de datos y bibliotecas digitales para encontrar trabajos representativos que aportaran a la investigación.

1. MÉTODO

Según [7, 8] una revisión sistemática de la literatura implica la realización de tres pasos básicos: 1) planificación, 2) realización, y 3) documentación, los cuales involucran seis procesos adicionales:

1. Definir las preguntas de investigación
2. Definir el proceso de búsqueda
3. Definir los criterios de inclusión y exclusión
4. Definir la valoración de la calidad
5. Definir la recopilación de datos
6. Definir el análisis de resultados

1.1 Preguntas de investigación

Las preguntas están orientadas a determinar qué tan importante es la documentación en la etapa de elicitación y cuál es su nivel de éxito en los proyectos. En esta investigación se plantearon:

PI1 ¿Cuál es el nivel de difusión acerca de cómo documentar la elicitación de requisitos?
PI2 ¿Qué tipo de trabajos se difunde?
PI3 ¿Cómo se difunde los trabajos?
PI4 ¿Cuál es nivel de éxito y el grado de aceptación y seguimiento en la comunidad?

1.2 Proceso de búsqueda

El objetivo de la búsqueda de información fue identificar los estudios que se ha publicado acerca de cómo documentar la elicitación de requisitos, para reconocer y resaltar fuentes específicas, estudios y opiniones candidatos a ser incluidos en la muestra final de trabajos. Para lograr este objetivo, en los parámetros de búsqueda se incluyeron las siguientes palabras clave: *elicitation documentation, elicitation models, elicitation approaches, elicitation methods, elicitation methodology, elicitation standard, elicitation techniques*. Estos términos debían aparecer en el título, el resumen o en el contenido del documento. La búsqueda se hizo en las bases de datos y bibliotecas digitales de IEEE Digital library, ACM Digital Library, Springer Link y Science Direct. Luego de aplicar el proceso se encontraron 73 trabajos que, en su mayoría se identificaron al buscar la palabra clave en el título o en el resumen. Para lograr un mayor cubrimiento no se excluyó ninguna fuente inicial.

1.3 Criterios de inclusión y exclusión

Para incluir los trabajos seleccionados como fuente primaria el contenido debía hacer un aporte importante al proceso de la documentación de la elicitación de requisitos, y de acuerdo con Dyba y Dingsoyr [9] existe cuatro fases para filtrar un documento desde una serie de trabajos al conjunto de datos primarios:

1. Identificar los estudios relevantes: la búsqueda arrojó 73 trabajos.
2. Excluir estudios con base en el título: se excluyeron 10 trabajos.
3. Excluir estudios con base en los resúmenes: se excluyeron 45 trabajos.
4. Seleccionar los más importantes para la temática en cuestión: 18 trabajos como muestra final. El criterio científico aplicado para esta selección consistió en valorar el aporte real que hace el trabajo a la documentación de la elicitación. En este caso se tuvo en cuenta si es teórico, experimental o un estudio de caso en la industria.

Durante la exploración se encontró poca información acerca de la temática, por lo que se decidió incluir investigaciones que en su contenido hicieran referencia a cómo documentar la elicitación de requisitos. Los criterios de inclusión y exclusión más importantes fueron: autoridad del autor, calidad y aporte del documento a la temática investigada, y claridad de la descripción.

1.4 Valoración de la calidad

Los estudios seleccionados son sólidos en cuanto a la calidad, lo cual garantiza la integridad de la información debido a que se soportan en la fuente de información primaria, una biblioteca digital o una base de datos con amplio reconocimiento en la comunidad, o en sitios web respaldados por una adecuada referenciación.

1.5 Recopilación de los datos

Luego de realizar el proceso de inclusión y exclusión se estructuró el conjunto de final de trabajos, y se determinó los atributos para los estudios primarios:

- Tipo de publicación
- Publicado en
- Editorial que respalda
- Año de publicación
- Clasificación del enfoque de investigación
- Clasificación del método de investigación

1.6 Análisis de resultados

Los resultados demuestran que la mayoría de fuentes hace referencia a la documentación de la especificación de requisitos, pero poco a la de la elicitación. En este proceso el objetivo era determinar:

1. Nivel de difusión acerca de cómo documentar la elicitación de requisitos: cantidad de trabajos encontrados.
2. Tipo de trabajo que se publica: opinión, descripción o investigación.
3. Cómo se difunden: libros, artículos, presentaciones en eventos, reportes técnicos.
4. Nivel de éxito y el grado de aceptación y utilización por parte de la comunidad relacionada: número de referencias que se realiza a los trabajos analizados, aceptación en la industria.

2. RESULTADOS

En la Tabla 1 se detalla los 18 trabajos de la muestra final de la revisión.

Tabla 1. Documentos incluidos en la muestra final.

ID	Documento
D1	Hurley R. (1982). Decision Tables in Software Engineering. Van Nostrand Reinhold.
D2	Jacobson I. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley.
D3	Parnas D. y Madey J. (1995). Functional Documents for Computer Systems. Science of Computer Programming 25(1), 41-61.
D4	Sommerville I. et al. (1998). Viewpoints for Requirements Elicitation: A Practical Approach. En 3rd International Conference on Requirements Engin. Putting Requirements Engineering to Practice. Colorado Springs, USA.
D5	Beck K. (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley.
D6	Berezin T. (1999). Writing a Software Requirements document. Documento propio.
D7	Cooper K. y Ito M. (2002). Formalizing a structured natural language requirements specification notation. En Twelfth Annual International Symposium of the International Council On Systems Engineering. Las Vegas, USA.
D8	Power N. y Moynihan T. (2003). A Theory of Requirements Documentation Situated in Practice. En 21st annual international conference on Documentation. San Francisco, USA.
D9	Mich L. et al. (2004). Market research for requirements analysis using linguistic tools. Requirements Engineering 9(2), 40-56.
D10	Smith S. et al. (2004). Requirements Analysis for Engineering Computation - A Systematic Approach for Improving Reliability. En Reliable Engineering Computing Workshop. Savannah, Georgia.
D11	Parnas D. (2006). From Requirements to Architecture. En H. Fujita (Ed.), New Trends in Software Methodologies, Tools and Techniques (pp. 3-36). IOS Press.
D12	Gallina B. et al. (2007). A Template for Requirement Elicitation Document of Software Product Lines. Technical Report. TR-LASSY-06-08. University of Luxembourg.
D13	Pleeger, S. y Atlee J. (2009). Software Engineering. Pearson.
D14	Laporti V. et al. (2009). Athena: A collaborative approach to requirements elicitation. Computers in Industry 60(6), 367-380.
D15	Crabtree C. et al. (2009). Exploring Language in Software Process Elicitation: A Grounded Theory Approach. En 3rd International Symposium on Empirical Software Engineering and Measurement. Lake Buena Vista, USA.
D16	Murtaza G. et al. (2010). A Framework for Eliciting Value Proposition from Stakeholders. WSEAS Transactions on Computers 9(6), 557-572.

D17	Menten A. et al. (2010). Using Audio and Collaboration Technologies for Distributed Requirements Elicitation and Documentation. En Third Inter. Workshop on Managing Requirements Knowledge. Sidney, Australia.
D18	Sajid A. et al. (2010). Modern Trends Towards Requirement Eliciation. En 2010 National Software Engineering Conference. Rawalpindi, Pakistan.

- D1 *Decision Tables in Software Engineering*. Las tablas propuestas proporcionan una notación que se traduce en acciones y condiciones en un formato tabular, y se pueden utilizar como una entrada legible por la máquina a un algoritmo basado en tablas. Según el autor la técnica es útil cuando se tiene un conjunto de condiciones complejo, y cuando las acciones se encuentran dentro de un componente, por ejemplo, en una elicitación. Lo que no tiene en cuenta es que generalmente las partes interesadas expresan sus necesidades en lenguaje natural, y estas tablas no permiten evaluar las ambigüedades resultantes antes de re-conocer esas acciones como requisitos. Esta propuesta está más orientada a crear un listado de candidatos a requisitos, por lo que no recibió una acogida aceptable en la comunidad de la Ingeniería del Software.

- D2 *Object-Oriented Software Engineering*. La representación gráfica compacta es útil para representar los requisitos, por lo que es recomendable utilizar diagramas de casos de uso para el modelado de los funcionales. Esta representación actúa como un puente entre los actores técnicos y el cliente, pero presenta desventajas y problemas, porque se aplica principalmente para modelar requisitos funcionales y no es muy útil para los no-funcionales. Además, carece de una semántica bien definida para realizar una adecuada documentación, lo que puede dar lugar a diferencias en la interpretación por las partes interesadas. A pesar de que la representación gráfica se utiliza como modelo de entendimiento entre las partes acerca del significado y descripción de las necesidades, como documento descriptivo de la elicitación todavía no logra brindar la información suficiente para las demás etapas de la Ingeniería de Requisitos, por lo que la propuesta se orienta más a brindar datos específicamente para la especificación, y poco para la documentación de la elicitación.

- D3 *Functional Documents for Computer Systems*. Un paso crítico al documentar los requisitos de un sistema es la identificación de los elementos ambientales, como las propiedades físicas –temperatura, presión−, los cuales, ya sea para medir o controlar, deben ser representados por variables matemáticas, y como es usual en ingeniería la asociación se debe definir con cuidado e indicarse sin ambigüedades. Para aclarar la asociación se puede utilizar diagramas entre los elementos y las variables matemáticas, como los presentados en esta propuesta. La cuestión es que para la mayoría de ingenieros el nivel matemático que exige es complejo, y no están preparados para utilizarla adecuadamente al momento de documentar la elicitación. Un problema que se acrecienta con los clientes y los usuarios, porque generalmente tienen menos capacidades de entendimiento acerca del contenido matemático de esta propuesta, por lo que no ha recibido buena aceptación en la comunidad en lo que tiene que ver con la documentación de la elicitación.

- D4 *Viewpoints for Requirements Elicitation: A Practical Approach*. Esta técnica soporta la agrupación, la gestión de requisitos, la verificación de inconsistencias, y la trazabilidad de los requisitos mediante puntos de vista. Parte de que, debido a los diferentes enfoques con los cuales las partes interesadas observan un requisito, es necesario recogerlos y organizarlos desde diferentes acercamientos. Esta técnica soporta la agrupación, la gestión de requisitos, la verificación de inconsistencias, y la trazabilidad de los requisitos. Pero esos puntos de vista reconocen múltiples perspectivas, lo que no proporcionan un marco para el descubrimiento de conflictos en los requisitos propuestos; además, no permiten la gestión de las

inconsistencias. Otra dificultad es que en un alto número de puntos de vista es difícil definir la prioridad de los requisitos, por lo que la documentación generada no es suficiente para solucionarlo.

- D5 *Extreme Programming Explained*. De acuerdo con el autor las historias de usuario se han utilizado como parte de la programación extrema y en las metodologías ágiles, y los clientes las pueden escribir utilizando una terminología no-técnica en el formato de frases y en lenguaje natural. Esta propuesta describe un proceso para la Ingeniería de Requisitos en el que el usuario tiene poca participación, y los formatos para documentarlos no tienen una estructura definida. Por otro lado, se deben expresar de forma no-gráfica y respetando una metodología específica, la cual no está en línea con paradigmas como UML y POO. Por todo esto la propuesta no ofrece una alternativa eficiente para documentar la elicitación, además, no tiene en cuenta a los requisitos no-funcionales, por lo que se deben recolectar y documentar separadamente.

- D6 *Writing a Software Requirements document*. Cada documento de requisitos consiste por lo menos de dos partes: una visión general y una descripción de la funcionalidad del sistema. A menudo debe incluir apéndices, de acuerdo a sí se necesita más información de la que contiene el resto del documento, o de material que no encaja en otra parte, pero que debe ser incluido. La dificultad radica en que el documento de la elicitación generado es muy extenso, y no ofrece una adecuada claridad acerca de la descripción de las necesidades de las partes interesadas. Por otro lado, se acoge más al estándar IEEE STD- 830-1998, que se orienta a la especificación, por lo que no es muy adecuada para documentar la elicitación.

- D7 *Formalizing a Structured Natural Language Requirements Specification Notation*. El autor propone una serie de estructuras con el objetivo de dar mayor organización a los documentos de la Ingeniería de Requisitos. Sin embargo, asume que los lenguajes naturales no tienen una notación formal ni gráfica específica, por lo que esas estructuras se pueden orientar tanto a algoritmos como a lenguajes de programación, lo que limita la libertad en la codificación posterior. El problema con esta propuesta radica en que las estructuras no son viables para documentar las descripciones de las necesidades, formuladas en lenguaje natural en la elicitación, por lo que las partes interesadas no pueden acordar las descripciones finales.

- D8 *A Theory of Requirements Documentation Situated in Practice*. Se trata de un marco teórico que intenta explicar la variedad de estilos para documentar los requisitos elicitados, en relación con la variedad de situaciones en las que el software se desarrolla. Este marco contrasta gran parte de la literatura sobre requisitos, porque tiende a acercarse a la documentación, pero abandona la situación de uso. La propuesta se divide en tres partes: 1) análisis de documentos de requisitos, como textos, para categorizar los diferentes elementos constitutivos que podrían ser utilizados para especificarlos; 2) esquema para clasificar el desarrollo del sistema con respecto al proceso de la documentación de requisitos; y 3) toma cada uno de estos tipos de situaciones y define un estilo apropiado para el documento de requisitos. Aunque puede ser útil y su objetivo es explicar las diversas formas en que en la práctica se documenta los requisitos, de acuerdo con los resultados al seguimiento de la propuesta esa diversidad no ha sido examinada por otro estudio, marco teórico o caso de estudio, lo que no permite conocer su eficiencia y efectividad en la práctica industrial.

- D9 *Market Research for Requirements Analysis Using Linguistic Tools*. En esta propuesta se describe que cuando la elicitación no presenta un modelo semánticamente transparente del dominio del problema, el analista debe utilizar objetos subyacentes, como archivos, líneas y caracteres. Con estos procedimientos se puede conseguir *buenos* requisitos, pero el proceso

de desarrollo suele ser lento. Para resolver este problema el autor propone que en la elicitación se tenga en cuenta diferentes características de los datos, como que son multilingües, se desarrollan de forma secuencial, se estructuran jerárquicamente, son multidimensionales, y tienen alta integración, por lo que se deben almacenar y seguir los vínculos asociativos entre las piezas de datos conexos. Pero si se tiene en cuenta estas características se incrementa la dificultad para documentar la elicitación, porque muchas de ellas todavía no se conocen en esta etapa. Aunque es una propuesta amplia, el documento que genera no pasa de ser una visión personal de cada ingeniero, en la que no se hace una adecuada discusión con las partes.

- D10 *Requirements Analysis for Engineering Computation - A Systematic Approach for Improving Reliability*. Esta propuesta utiliza tablas para representar los requisitos mediante un sistema de trazabilidad, las cuales incluyen las propiedades y las relaciones de los mismos. Esto ayuda para su organización y para mostrar explícitamente los diversos tipos de relaciones entre ellos. El diagrama es útil para estandarizar la forma de especificar los requisitos a través de una semántica definida, y permite su representación y de los elementos del modelo, lo que significa que los visiona como parte de la arquitectura del sistema. El problema surge cuando el cliente realiza modificaciones reiterativas, porque la estructura de las tablas no es flexible y no permite re-procesos para realizar los cambios necesarios. Esta falta de dinamismo no le brinda maniobrabilidad al ingeniero como para mantener actualizado el documento de la elicitación, por lo que la propuesta no ha tenido un seguimiento adecuado.

- D11 *From Requirements to Architecture*. La propuesta consiste de un modelo de dos variables, común en la ingeniería mecánica y en la eléctrica, que el autor utiliza para documentar los requisitos de esos sistemas. Aunque en teoría es útil para sistemas software, debido a la volatilidad de los mismos, no es práctico para documentar la mayoría de requisitos. Además, el modelo no es fácilmente adaptable a contextos en los que la volatilidad de las necesidades del cliente es alta, y donde se requiere mayor dinamismo para mantener actualizado el documento de la elicitación.

- D12 *A Template for Requirement Elicitation Document of Software Product Lines*. Este trabajo proporciona plantillas para describir informalmente las líneas de productos software, y para documentar su uso. El objetivo es que en la descripción se utilice un lenguaje natural comprensible para todos interesados. La primera plantilla se compone de un diccionario de datos en forma de tabla, y en la segunda se elicita los potenciales requisitos funcionales y no-funcionales. Cada caso de uso involucrado proporciona uno o más escenarios, que describe cómo debe interactuar el sistema con el usuario final, o con otro sistema, para lograr los objetivos específicos del negocio. Pero esa interacción no se refleja en las tablas, por lo que cualquier modificación en los requisitos genera una nueva tabla de datos.

- D13 *Capturing the Requirements*. Los requisitos son inciertos, lo que hace difícil emplearlos en procesos donde se debe actualizar los modelos cada vez que cambian. Como una alternativa a este problema los métodos ágiles reúnen y aplican los requisitos en incrementos. El proceso consiste en partir de una versión inicial, en la que, de acuerdo con la definición de los objetivos del negocio de las partes interesadas, se obtiene la mayoría de requisitos esenciales; o con el uso del sistema o con una mejor comprensión del problema, cuando surgen nuevos requisitos. Este desarrollo incremental permite una entrega temprana y continua, y tiene la capacidad emergente de obtener requisitos de última hora. La cuestión es que los procesos ágiles para requisitos, en los que el sistema se construye de acuerdo con los que se van definiendo en el momento, no tienen en cuenta la planificación o el diseño para posibles requisitos futuros, y la documentación se queda corta en todo paso.

- D14 *Athena: A collaborative approach to requirements elicitation Link*. La elicitación de requisitos es una fase que requiere mucha interacción entre las partes interesadas, porque se necesita un alto intercambio de información, y muchas veces las técnicas no son usadas de forma correcta. Además, los clientes no están completamente seguros de sus necesidades reales, lo que lleva a que los requisitos sean incompletos e inconsistentes. Esos problemas de comunicación entre las partes suceden porque los clientes utilizan el lenguaje natural para expresarse, mientras que los analistas prefieren uno más formal. Athenea es una herramienta para que los usuarios plasmen sus experiencias y puntos de vista, para luego negociar los requisitos de forma colaborativa. A partir de descripciones de alto nivel adopta un enfoque basado en un perfeccionamiento gradual de requisitos y termina con casos de uso concretos, lo que se logra de forma cooperativa en el equipo, y al final se traduce en requisitos negociados. La cuestión es que es una herramienta experimental, y luego de años de divulgada todavía no ha sido probada en entornos reales de desarrollo de software, por lo que no es posible verificar su eficiencia y efectividad para generar el documento de la elicitación.

- D15 *Exploring Language in Software Process Elicitation: A Grounded Theory Approach*. La elicitación es un paso importante para la creación del documento que describe los requisitos. En este proceso se requiere actividades de comunicación, de extracción de conocimiento y de comprensión de los objetivos, con la intención de elaborar un documento que demuestre el cumplimiento de políticas de auditoria y la aplicación de buenas prácticas, para identificar cambios potenciales en el flujo de trabajo existente. La experimentación descrita en este trabajo se divide en dos momentos: 1) escuchar a los usuarios cuando expresan sus necesidades en lenguaje natural, y 2) utilizar plantillas para documentar el proceso. Los resultados incluyen representaciones del proceso, notas del campo de observación y transcripciones de entrevistas, pero al momento de analizarlas se encuentra con que se sugieren diferentes formas para que los usuarios del proceso describan sus necesidades, lo que dificulta la documentación esperada.

- D16 *A Framework for Eliciting Value Proposition from Stakeholders*. Teniendo en cuenta la importancia de las partes interesadas en el proceso, el objetivo de esta propuesta es simplificar y estructurar el proceso de elicitación de requisitos, y generar un documento que exponga una serie de criterios para su selección, como la influencia, la importancia dentro del proyecto, la legitimidad, la urgencia y el interés en el proyecto. Aunque tiene relevancia, solo se encontraron trabajos en los que se aplica para documentar requisitos en proyectos a pequeña escala, por lo que todavía no se puede considerar su éxito en proyectos de mayor tamaño. La desventaja de esto es que los problemas que actualmente soluciona la Ingeniería del Software son grandes, complejos y complicados, y con una propuesta de este tipo no se podría generar un documento de elicitación que brinde agilidad para pasar rápidamente al diseño.

- D17 *Using Audio and Collaboration Technologies for Distributed Requirements Elicitation and Documentation*. En este trabajo se presenta un método para elicitar y documentar requisitos que utiliza tecnologías de colaboración y grabaciones de audio, para permitirles a las partes interesadas la elicitación y la documentación conjunta. Esta documentación se puede realizar mediante colaboración entre todas las partes, mientras que el control de cambios en la trazabilidad se realiza mediante el control de versiones integradas. Para estructurar la documentación de una plantilla previamente desarrollada se puede utilizar términos clave capturados y definidos en un glosario, pero, aunque estos términos son útiles para establecer un lenguaje común y para evitar posibles ambigüedades, la relación posterior con los términos utilizados en la documentación no es clara y se dificulta hacerle trazabilidad.

- D18 *Modern Trends Towards Requirement Elicitation*. De acuerdo con el autor la Ingeniería de Requisitos es un grupo de actividades que ayuda a encontrar y comunicar las necesidades, el propósito y el contexto de un sistema. Este proceso se inicia con la elicitación de los requisitos, a lo que considera como el principal factor de fracaso de la mayoría de proyectos software. En la propuesta presenta una forma para analizar los datos elicitados, que se obtienen a partir de la información y la documentación de los requisitos del sistema, y desde la cual se extrae las directrices modeladas mediante la aplicación de diferentes técnicas. También propone un plan para elicitar requisitos, que pretende superar las limitaciones que enfrentan los ingenieros al momento de documentar lo elicitado. Aunque el objetivo de la documentación es claro, deja a las demás partes interesadas sin un rol claro, por lo que la versión final es una visión solo para y desde los ingenieros, sin tener en cuenta el diálogo que se necesita con los clientes y usuarios.

2.1 Respuesta a las preguntas de investigación

2.1.1 PI1 ¿Cuál es el nivel de difusión acerca de cómo documentar la elicitación?

Luego de aplicar el proceso de búsqueda en las diferentes bases de datos se encontraron 73 trabajos que contenían, por lo menos, una de las palabras clave seleccionadas. Un número que puede considerarse reducido dada la cantidad de publicaciones que año tras año se suman a los contenidos de estas bases de datos. Esto valida la hipótesis de la investigación en el sentido de que la comunidad está trabajando poco acerca de cómo documentar la elicitación de requisitos. Cerca del 75% contiene solo alguna de las palabras clave y en el cuerpo del documento no trasciende como aporte significativo, ni siquiera como campo para futuros trabajos. Solo alrededor del 25% de los trabajos (18 en total) contiene alguna descripción o sugerencia al proceso de la documentación de la elicitación.

2.1.2 PI2 ¿Qué tipo de trabajos se difunde?

En la muestra se propone *frameworks* como conjunto estandarizado de prácticas para enfocar la documentación; también se habla de herramientas software que pueden ayudar a mejorar el proceso, como ATLAS [D16], que permite analizar los datos empíricos obtenidos de las entrevistas, y ATHENEA [D14], que les permite a los usuarios registrar sus experiencias y puntos de vista, y negociar los requisitos de forma colaborativa. En cuanto al tipo de trabajos, para esta revisión se determinaron las variables de opinión, reflexión e investigación (Tabla 2).

Tabla 2. Tipo de trabajos publicados

Tipo	Cantidad	
Opinión	29	39,72%
Reflexión	26	35,61%
Investigación	18	24,65%

De acuerdo con estos datos se puede concluir que la mayoría de investigadores se orienta por la opinión y la reflexión, y en menor proporción por la investigación en este tema. Esta baja participación se puede deber a que este tema es una cuestión que apenas comienza a preocupar a la comunidad, y a que los investigadores la empiezan a percibir como un hito importante para el mejoramiento de la calidad del desarrollo de software. Algunos autores argumentan que este tema es un problema en la mayoría de los proyectos actuales, en parte porque las empresas estructuran sus propios formatos y metodologías para documentar su proceso de elicitación. Pero esta forma de trabajar no es la más recomendada, porque las necesidades son diferentes y los contextos también, entonces toma importancia la necesidad de un estándar para documentar la etapa de elicitación.

2.1.3 PI3 ¿Cómo se difunde los trabajos?

Para responder a esta pregunta solo se analizaron los trabajos que hacen énfasis en investigación, porque el objetivo era encontrar de qué forma se están difundiendo propuestas que busquen solucionar el problema de la documentación de la elicitación. La mayoría son artículos publicados en revistas que se encuentran en las bases de datos utilizadas, pero también se hallaron trabajos en congresos, libros y reportes técnicos de organismos internacionales. Se esperaba que los artículos fueran el medio más utilizado para difundir las investigaciones, pero son los eventos los que tienen mayor acogida por los investigadores, como se observa en la Tabla 3.

Tabla 3. Cómo se difunden los trabajos

Medio	Cantidad	
Revistas	3	16,7%
Congresos	7	38,9%
Libros	4	22,2%
Reportes técnicos y otros	4	22,2%

Estos resultados se pueden explicar desde el punto de vista de que un congreso es un encuentro en el que se puede discutir directamente con la comunidad, y se retroalimenta más rápidamente. De esta forma los investigadores pueden complementar pronto su trabajo o darle un nuevo giro para luego experimentar nuevamente.

2.1.4 PI4 ¿Cuál es nivel de éxito y el grado de aceptación y seguimiento en la comunidad?

Para verificar el grado de aceptación de las propuestas en la comunidad, se realizó una búsqueda de trabajos que citan los documentos de la muestra final. El proceso se realizó en las mismas bases de datos que se utilizaron para la revisión. Teniendo en cuenta que, luego de ser difundida, una propuesta necesita entre cinco y siete años para lograr un adecuado nivel de aceptación y de difusión en la comunidad, y que se acepta que un nivel es aceptable si el trabajo ha obtenido en ese lapso de tiempo por lo menos cuatro referencias en medios de igual o mayor impacto que el original [10], en la Tabla 4 se muestra los resultados que arrojó la búsqueda para responder a esta pregunta.

Tabla 4. Nivel de aceptación

ID	Referenciado	Nivel	
D5	316	Alto	22,3%
D2	309	Alto	
D3	94	Alto	
D4	24	Alto	
D8	10	Bueno	11,1%
D1	9	Bueno	
D11	5	Aceptable	22,2%
D14	5	Aceptable	
D15	5	Aceptable	
D10	4	Aceptable	
D9	3	Bajo	16,7%
D12	2	Bajo	
D17	1	Bajo	
D6	0	Nulo	27,7%
D7	0	Nulo	
D13	0	Nulo	
D16	0	Nulo	
D18	0	Nulo	

Estos resultados ratifican lo expresado acerca del tiempo que la comunidad se toma para comprender, utilizar, aceptar y referenciar una propuesta. D5, D2, D3 y D4, que poseen el mayor nivel de aceptación, son trabajos de los años 90, por lo que han permanecido el tiempo suficiente para lograr impactar a los demás investigadores; mientras que D16 y D18 están llegando al tiempo necesario de comprensión y posiblemente de referenciación. En términos generales se puede concluir que el nivel de aceptación y seguimiento a estos trabajos es bueno, debido a que 10 de los 18 (55.6%) documentos de la muestra tienen un nivel entre Aceptable y Alto.

En cuanto al nivel de éxito en la aplicación en la industria se encontró que D2 y D4 son las propuestas que más se ha experimentado, y que tienen un reporte de aceptación Bueno. El hecho de que sean las más aplicadas se puede explicar por el alto reconocimiento que tienen sus autores. Este nivel en el reporte no significa que sean las de mayor éxito, porque también se encontraron otros en los que se describen problemas de acoplamiento y de comprensión al momento de aplicarlas. En cuanto a ATLAS y ATHENA, aunque son *framerworks* que no requieren demasiado esfuerzo para aplicar y experimentar, no se encontraron reportes que permitiera evaluar su nivel de éxito.

3. ANÁLISIS DE RESULTADOS

Aunque en los documentos que conforman la muestra los autores aceptan y asumen a la Ingeniería de Requisitos como la fase más importante del ciclo de vida de la Ingeniería del Software, y a que brindan especial cuidado a la elicitación, llama la atención el hecho de que trabajen, propongan y difundan muy poco acerca de cómo documentar esta etapa. La mayoría se limita a expresar que se debe documentar con seriedad, claridad y formato, para lo cual proponen algunas estructuras, pero, de la muestra, prácticamente ninguno describe la forma para lograr un documento con estas características. Documentar la elicitación de requisitos es importante, porque este documento es la guía sobre la que se desarrollan las demás etapas de la Ingeniería de Requisitos, que a su vez es la base para realizar las demás fases del ciclo de vida del producto.

Las estructuras sugeridas pasan por tablas, gráficas, casos de uso, puntos de vista, escenarios y hasta pistas de audio, que se proponen como base para el documento de esta etapa. Pero, debido a la complejidad de los problemas actuales que se resuelven con productos software, esto no es suficiente, porque se requiere un modelo matemáticamente estructurado para crear un documento formal, o semi-formal, cuya información permita resolver las ambigüedades de los requisitos, con la flexibilidad suficiente para incorporar las modificaciones y actualizaciones que se presentan en la elicitación. Esto no se plantea, ni se describe en los trabajos analizados en esta revisión, y se debe a que la mayoría apunta a la documentación de la especificación, la etapa final de la Ingeniería de Requisitos.

Por otro lado, los documentos que conforman la muestra final se orientan a estudios exploratorios y de trabajo en grupos interdisciplinarios, cuyos resultados hacen evidente la necesidad de un estándar para orientar la documentación de la elicitación. En parte porque los investigadores se han dado cuenta de que sus formatos no funcionan en todos los contextos, por lo que cada vez se deben re-estructurar.

Así mismo, llama la atención que pocos trabajos de la muestra final tienen en cuenta a los métodos formales (D3, D6, D8, D18), mientras que la mayoría se basa en un proceso de elicitación teniendo como referente al lenguaje natural (D1, D2, D4, D5, D7, D9-D17). Esta práctica genera uno de los problemas de mayor impacto en la Ingeniería de Requisitos: la ambigüedad de la descripción, que hace que las partes interesadas no puedan clarificar y unificar sus propias

necesidades. Lo que sí queda demostrado es que la mayoría de propuestas pretende pasar directamente a la especificación, sin tener un adecuado documento de la elicitación.

4. CONCLUSIONES

Este capítulo es el resultado de una revisión sistemática de la literatura acerca de las propuestas para documentar la elicitación de requisitos, y pretende ser un compendio del estado actual en la investigación relacionada. Para lograrlo se realizó una búsqueda en las bases de datos, para recopilar los trabajos que relacionaran esta temática. Aunque la muestra inicial fue amplia (73), en el proceso de inclusión y exclusión aplicado se conformó una muestra final de 18. Esto se debe a que la mayoría de los no-incluidos apunta a la documentación de la especificación, la etapa final de la Ingeniería de Requisitos.

Por otro lado, los estudios que conforman la muestra final se orientan a estudios exploratorios y de trabajo en grupos interdisciplinarios, cuyos resultados hacen evidente la necesidad de un estándar para orientar la documentación de la elicitación. En parte, porque los investigadores se han dado cuenta de que sus formatos funcionan en un contexto, pero para otro no, por lo que deben re-estructurarlo cada vez.

También llama la atención que pocos trabajos de la muestra final tienen en cuenta a los métodos formales, mientras que la mayoría menciona o se basa en un proceso de elicitación en lenguaje natural. Este ejercicio genera uno de los problemas de mayor impacto en la Ingeniería de Requisitos: la ambigüedad de la descripción, que hacen las partes interesadas acerca de sus necesidades.

Aunque en la mayoría de estudios analizados se describe a la documentación de la elicitación como un proceso importante y de valor para las demás actividades de la Ingeniería de Requisitos, estos principios no se tienen en cuenta, porque la mayoría pretende pasar directamente a la especificación, sin tener un documento adecuado de la elicitación. Esto se evidencia en los diferentes re-procesos que deben realizar los ingenieros en el diseño del producto.

De forma general, la investigación desde la que se origina este trabajo tiene como propósito estructurar un modelo semi-formal para documentar la elicitación de requisitos, y de acuerdo con los resultados encontrados en esta revisión se necesita de forma acelerada. Porque mientras no se tenga una propuesta estructurada y genérica esta documentación se seguirá pasando por alto, ocasionando que la especificación no sea la adecuada para continuar con las demás fases del ciclo de vida del producto.

REFERENCIAS

[1] Davis A. (1993). Software Requirements: Objects, functions, and states. Prentice-Hall.

[2] Serna E. et al. (2012). SEDLO: Software Engineering for developing learning objects. En 6th Euro American Conference on Telematics and Information Systems. Valencia, España.

[3] Loucopoulos P. y Karakostas V. (1995). System Requirements Engineering. McGraw-Hill.

[4] Mustelier D. y Viera Y. (2013). Variables that define the complexity of the software functional requirements. Revista Antioqueña de las Ciencias Computacionales y la Ing. de Software 3(2), 38-42.

[5] Christel M. y Kang K. (1992). Issues in Requirements Elicitation. Technical Report CMU/SEI-92-TR-012. Carnegie Mellon University.

[6] Serna E. (2010). Formal methods and Software Engineering. Revista Virtual Universidad Católica del Norte 30, 158-184.

[7] Kitchenham B. (2004). Procedures for undertaking systematic literature reviews. Technical Report. Keele University.
[8] Kitchenham B. et al. (2009). Systematic literature reviews in Software Engineering: A systematic literature review. Information and Software Technology 51(1), 7-15.
[9] Dyba T. y Dingsoyr T. (2008). Empirical studies of agile software development: A systematic review. Information and Software Technology 50(9-10), 833-859.
[10] Serna E. (2012). Current State of Research on Non-Functional Requirements. Revista Ingeniería y Universidad 16(1), 225-246.

CAPÍTULO XVIII
Un modelo para documentar la elicitación de requisitos

Edgar Serna M.[1]
Jorge Hernán Suaza J.[2]
[1]Instituto Antioqueño de Investigación
[2]Instituto Metropolitano de Educación

Se realizó una revisión sistemática de la literatura para determinar la vigencia y efectividad de los modelos para documentar la elicitación de requisitos, y se pudo determinar que los que se propone actualmente se centran en técnicas para colectar información, pero le prestan poca atención a la documentación. Además, se basan fuertemente en el lenguaje natural, por lo que se dificulta su interpretación y, debido a las ambigüedades, se genera re-procesos en las etapas posteriores del ciclo de vida. Al realizar el análisis a los resultados de la revisión se concluyó que es posible, y se requiere, tomar las mejores prácticas documentadas y adicionarles principios desde la lógica, la abstracción, y los métodos formales para estructurar un modelo semi-formal para documentar la elicitación. En este capítulo se describe el modelo REDOC y su aplicación y validación en la comparación con cinco de los modelos encontrados en la revisión.

INTRODUCCIÓN

El software se diseña y desarrolla a través de una serie de etapas conocidas como ciclo de vida, con las que se estructura y desarrolla una solución a las necesidades del cliente. Es una labor de trabajo en equipo entre diversos actores, en la que la mayor parte de los inconvenientes en el proceso se relacionan con la Ingeniería de Requisitos, y más específicamente en la etapa de elicitación. Entonces, si las necesidades del cliente no se elicitan adecuadamente el producto presentará problemas, porque no responderá a las necesidades establecidas. Es aquí donde radica la importancia de la elicitación, porque en ella se genera la información necesaria para la especificación, que a su vez es la base para el diseño y el desarrollo de la solución.

Por otro lado, la elicitación debe responder a las necesidades de la Ingeniería del Software, porque los problemas que debe resolver son complejos, y porque se requiere procesos que minimicen la re-ingeniería y agilicen la transición al diseño [1]. Pero el alcance del sistema, la falta de comprensión del problema y la volatilidad de los requisitos incrementan la complejidad del proceso. Es por esto que las diferentes etapas de la Ingeniería de Requisitos son exigentes e implican procesos sistemáticos e iterativos en los que el analista debe recurrir a la lógica, a la abstracción y a sus capacidades de modelado, con el objetivo de hacer una representación mental del problema e iniciar el proceso de solución.

Pero actualmente los métodos que se utilizan son informales y se basan en el lenguaje natural, lo que impregna mayor complejidad al proceso de la Ingeniería del Software. En tales circunstancias se requiere formalizar parte o todo el proceso, y de esta manera alcanzar una mejor interpretación de las necesidades del cliente.

Un método es formal si posee una base matemática estable, normalmente soportada por un lenguaje de especificación, que permita definir de manera precisa nociones como consistencia, completitud, especificación, implementación y correctitud [2]. Al utilizar notaciones y lenguajes formales es posible estructurar claramente los requisitos del sistema, y generar las especificaciones que permitan definir su comportamiento de acuerdo con *qué* debe hacer y no con el *cómo* se hace [3]. Pero esto estará incompleto sino no se parte de una sólida documentación de los requisitos elicitados, lo que agiliza los procedimientos en las demás etapas y sustenta la comprensión del problema a solucionar. La propuesta de diseñar una modelo semi-formal para documentar la elicitación de requisitos parte de incluir a la descripción textual elementos formales, y así disponer de la información necesaria para reducir los problemas de comprensión de las necesidades del cliente y para el mejoramiento de la especificación.

Para llegar a esta conclusión, primero se realizó una revisión sistemática de la literatura para analizar la vigencia y efectividad de los modelos para elicitar requisitos desde la perspectiva de la documentación. Con las mejores prácticas documentadas en ellos, y con la adición de principios desde la lógica y los métodos formales, se estructura un modelo semi-formal para documentar la elicitación de requisitos, y como respuesta a las necesidades de la Ingeniería del Software actual, que no alcanza a satisfacer los modelos estudiados.

1. MÉTODO

La metodología utilizada es una revisión sistemática de la literatura aplicando la propuesta del profesor Serna [4], en la que se buscaron estándares y normas para documentar la elicitación de requisitos, y se revisaron las propuestas que se divulga con el mismo objetivo.

2. RESULTADOS Y ANÁLISIS

2.1 Antecedentes

La construcción de un Sistema de Información es una tarea difícil por los problemas que se presenta en la traducción de los requisitos en un proyecto software. En la Ingeniería de Requisitos, como primera fase de la Ingeniería del Software, se debe identificar, definir y documentar las necesidades para satisfacer adecuadamente los objetivos del sistema. Pero, aunque se ha propuesto diferentes técnicas para elicitar requisitos [5], y su aplicación permite comprender dichas necesidades, la mayoría no describe un modelo para documentar y garantizar que la elicitación sea visible y comprendida por las partes interesadas.

Para lograr este objetivo los requisitos se deben documentar adecuadamente, una actividad importante para asegurar que las partes lleguen a acuerdos comunes acerca de lo que requieren modelar y presentar como solución al problema [6]. En la revisión no fue posible identificar un estándar que proporcione una orientación o procedimiento directamente relacionado a cómo documentar la elicitación de requisitos, sin embargo, diversos procesos, normas, estándares e iniciativas utilizan esta etapa como el punto de partida para iniciar un proyecto software. Entre ellos se puede citar:

1. *Project Management Body of Knowledge PMBOK* [7]. Un conjunto de prácticas de gestión que sirven de base para la metodología de gestión de proyectos. PMBOK define la elicitación de requisitos como el proceso en el que se define y documenta las necesidades de las partes interesadas a través de la iniciación, la planeación, la ejecución, el monitoreo y el control y cierre. Pero, aunque tiene amplia acogida en la Ingeniería del Software, no describe qué y cómo documentar esta etapa.

2. *Capability Maturity Model Integration CMMI* [8]. Una serie de prácticas y procesos de alto nivel que les ayudan a las organizaciones a edificar un modelo para mejorar sus procesos, y en cierta medida la calidad de los mismos. El desarrollo de requisitos se enmarca en el nivel 3 de CMMI y su propósito es identificar, determinar y analizar las necesidades del cliente. Pero, aunque este nivel describe el procedimiento a seguir mediante 13 actividades, ninguna está orientada a documentar los requisitos elicitados.

3. *Software Engineering Body of Knowledge SWEBOK* [9]. Definido como una guía al conocimiento para la Ingeniería del Software. Específicamente en requisitos define los procesos de análisis de necesidades del cliente para descubrir y resolver conflictos, descubrir falencias y cómo interactuar con el contexto. Aunque la guía tiene en cuenta que no realizar bien esta fase trae consecuencias graves en el desarrollo del producto software, no se refiere explícitamente ni describe un procedimiento para documentar la elicitación.

4. Existe otras normas y estándares relacionados, tales como: ISO/IEC 12207 Information Technology – Software Life-Cycle Processes; ISO/IEC TR 15504 Software Process Assessment; IEEE STD- 830-1998 Recommended Practice for Software Requirements Specifications; MIL-STD 490A Specification Practices; MIL-STD 498 Software Development and Documentation; SA PSS-05 European Space Agency; ISO/IEC 9126 Information Technology - Software Product Evaluation - Quality characteristics and guidelines for their use; CMU/SEI-92-TR-012; y CMU/SEI-2006-TR-013, entre otras. Si bien todas asumen que la documentación de los requisitos es esencial para dinamizar los procesos del ciclo de vida, ninguna describe o propone un modelo, método o proceso para hacerlo.

Además de los antecedentes descritos, varios autores han propuesto y divulgado modelos para elicitar requisitos. El análisis de esas propuestas fue desarrollado por Serna y Suaza, cuyos resultados hacen parte de este libro. En esa revisión de la literatura encontraron que los autores describen a la documentación de la elicitación como un proceso importante y de valor para las demás actividades de la Ingeniería de Requisitos, pero no tienen en estos principios, porque pretenden pasar directamente a la especificación, sin tener un documento adecuado de la elicitación. Esto se evidencia en los diferentes re-procesos que deben realizar los ingenieros en el diseño del producto.

2.2 Mejores prácticas

Debido a que en esa revisión no se pudo encontrar un modelo que permitiera documentar formal o semi-formalmente la elicitación de requisitos, en esta investigación se optó por seleccionar una serie de conceptos, considerados o no por los investigadores, como necesarios para un modelo de este tipo.

- *Plantillas*. Se utilizan como un medio para colectar información y documentar procesos. En la elicitación son una herramienta útil para almacenar los datos necesarios para identificar y reconocer las necesidades del cliente. Algunos autores las utilizan como modelos de elicitación y negociación, y las aplican como complemento a las técnicas de elicitación, donde la documentación final se maneja directamente por la interacción de los participantes.

- *Esquemas*. En la práctica muchos trabajos proponen la elaboración de diversos tipos de diagramas y esquemas para observar factores de la elicitación de forma organizada. Otros utilizan esquemas conceptuales orientados a objetos que se obtienen a partir del modelo de especificación. Las herramientas CASE son otra alternativa utilizada especialmente para detallar los requisitos y originar esquemas. En general, los autores los utilizan para representar sub-sistemas o componentes de un sistema, o las relaciones entre ellos.

- *Matrices de variables*. Se utilizan como una forma analítica de hacerle frente a la interacción de los factores en la capa inicial de la lógica de un programa. Para algunos autores las variables que interactúan, y el comportamiento del sistema, solo serán correctos cuando se documenten apropiadamente. En general, se utilizan como un complemento de las plantillas, porque también usan variables o campos para asociar y dividir apropiadamente la información.

- *Indicadores*. El uso de indicadores permite evaluar de qué forma la elicitación contribuye a mejorar la interpretación del problema. La visión orientada a procesos que algunos aplican exige definir las actividades que componen el proceso sistemáticamente, identificar la interrelación entre ellas y los responsables, e introducir indicadores para medir los resultados de capacidad y eficacia del mismo. Como consecuencia deben introducir criterios que les permita mejorar el proceso de la elicitación.

- *Escenarios*. Algunos autores utilizan escenarios para describir el comportamiento del sistema y asegurar una mejor comprensión y colaboración entre las partes en el proceso de elicitación de requisitos, y para mantener información que pueden reconocer. El uso de esta práctica en la elicitación permite la validación de los requisitos con los usuarios. El propósito principal en la literatura es capturar el vocabulario de la aplicación y su semántica, para facilitar la comprensión de la funcionalidad de la aplicación, porque cada escenario describe una situación específica de la aplicación, centrando la atención en su comportamiento.

- *Diagramas*. Algunos trabajos plasman los requisitos en diagramas para mostrar la interacción de los diversos actores con el sistema, lo cual añade valor para el usuario, como en los diagramas de casos de uso en los que especifican el comportamiento del sistema. Es decir, describen qué hace el sistema, no cómo lo hace. Esta práctica tiene la ventaja de indicar la secuencia del proceso en cuestión, las unidades involucradas y los responsables de su ejecución, es decir, son la representación simbólica o pictórica de un procedimiento administrativo.

2.3 Principios

- *Lógica y abstracción*. Su utilidad radica en que permiten comprender, analizar y modelar, tanto el problema como la posible solución, a la vez que les facilita a los usuarios y clientes la comprensión de lo que están interpretando, por ejemplo, los requisitos. Para esta investigación la lógica se concibe como un conjunto de principios que les permiten a las personas hacer juicios con base a evidencias, con el propósito de tomar decisiones sustentadas en el desarrollo de alguna actividad. Por su parte, la abstracción es el proceso de quitar o extraer características de algo con el fin de reducirlo a un conjunto de particularidades esenciales. En la elicitación de requisitos se pueden utilizar para escoger características comunes de las necesidades, objetos y procedimientos de los usuarios; o cuando dos funciones realizan casi la misma tarea y se pueden combinar en una sola función. Para estructurar el modelo semi-formal que se propone a partir de esta investigación se tiene en cuenta la comprensión, interpretación, modelamiento y resolución de problemas, y el razonamiento lógico. Ambos son principios necesarios para comprender el contexto y el problema mismo que se desea resolver, y como herramientas que facilitan encontrar la información necesaria para documentar la elicitación.

- *Métodos formales*. Se refieren a técnicas y herramientas basadas en principios y postulados matemáticos que se utilizan para especificar, diseñar, validar y verificar sistemas software y hardware, entre otros. La especificación utilizada en los métodos formales está conformada por enunciados bien formados en una lógica matemática. La verificación formal por deducciones rigurosas es la misma lógica, es decir, cada paso sigue una regla de inferencia y por lo tanto se puede comprobar mediante un proceso mecánico [10]. En la fase de Ingeniería de Requisitos, especificar formalmente es importante y es una labor que requiere mayor cuidado, porque su función es garantizar que tanto el funcionamiento como el desempeño del programa sean correctos, bajo cualquier situación. Los principios de los métodos formales que se tienen en cuenta en la estructuración del modelo son: el cálculo proposicional [11], las tablas de decisión [12], la teoría de conjuntos [12], los lenguajes declarativos [13], y el diseño por contrato [14].

3. MODELO PARA DOCUMENTAR LA ELICITACIÓN DE REQUISITOS

Semi-formal se entiende como una agrupación de códigos de procedimiento que indica el tipo de descripción utilizada para documentar la información [15], que se centra en la creación de un modelo de sistema en una etapa determinada del ciclo de vida de desarrollo, y su objetivo es realizar transformaciones de modelos automáticos [16].

A diferencia de un modelo no-formal, sus notaciones son intuitivas, permiten una mejor abstracción de los detalles y aplican metodologías estandarizadas y bien definidas [17]. Los modelos semi-formales son ampliamente utilizados en la industria del software debido a que su semántica ayuda a evitar la ambigüedad, la inconsecuencia y la imprecisión, y a que es razonada formalmente [18].

3.1 Estructuración

- Paso 1. *Elaborar el diagrama de procesos*. Un diagrama de procesos se utiliza para mostrar las relaciones entre los principales componentes de un sistema, también para tabular valores de diseño de procesos para los componentes en diferentes modos de funcionamiento. Muestra la relación entre los sub-procesos del sistema interpretado, pero oculta los detalles de menor importancia, como responsabilidades y denominaciones de agentes. Como representación gráfica, en REDOC se utiliza para visualizar el flujo de procesos, donde es posible observar las diferentes actividades del sistema y las conexiones entre ellas. Esto les permite a las partes interesadas una mejor comprensión del problema, y analizar la abstracción y el modelado de la solución.

- Paso 2. *Aplicar cálculo proposicional*. Es un principio de los métodos formales que se utiliza para formalizar el lenguaje natural, con el que los usuarios expresan sus necesidades, para estructurarlas en forma de proposiciones lógicas [15]. Las proposiciones se muestran en la lógica como objetos de un lenguaje formal mediante diferentes tipos de símbolos, que se concatenan de acuerdo con reglas recursivas para construir cadenas a las que se asigna valores de verdad. Al representar las necesidades de esta forma es posible verificar que las interacciones de los actores y el sistema respetan las reglas del negocio, y que en las actividades de prueba posteriores se obtendrá los valores esperados de entrada y de salida. En la elicitación de requisitos es importante para representar las necesidades como fórmulas matemáticas, además, para facilita la estructuración del plan de pruebas.

- Paso 3. *Elaborar el diagrama de caminos*. Un diagrama de caminos es la representación del modelo ideal de comportamiento del sistema por medio de un grafo dirigido. Para este proceso se asigna un valor secuencial de nodo y una prioridad y dependencia de ejecución, de acuerdo con el diagrama de procesos, y en sus aristas se detalla las pre y post condiciones. Este grafo se convierte en una vista de la información de partida y facilidad la construcción, lectura e interpretación del documento de la elicitación. Las proposiciones iteradas y representadas en el diagrama de caminos representan las acciones que los actores realizan sobre el sistema.

- Paso 4. *Completar la plantilla de la elicitación*. Las proposiciones iteradas y representadas en el diagrama de caminos representan acciones que un actor realiza sobre el sistema; un proceso que se lleva a cabo mediante una serie de interacciones, conformadas igualmente por una serie de acciones, que el sistema ejecuta en respuesta a las solicitudes del actor. Este proceso de comunicación produce un cambio de estado en las variables, los valores, las bases de datos y el hardware, que también se debe describir. Para documentar este proceso de comunicación actor-sistema-actor REDOC propone una plantilla para capturar la información del proceso. La plantilla contiene la información necesaria para generar el modelo de comportamiento del sistema: 1) *acción actor*, para detallar la acción que el actor realiza como parte de una interacción; 2) *reglas del negocio*, para verificar las operaciones, si los valores resultantes son los esperados, y que los casos de prueba no violen las reglas establecidas; 3) *acción/respuesta sistema*, para describir las acciones o respuestas que el sistema ejecuta, como acciones internas o como respuesta a una acción previa del actor; y 4) *cambio de estado*, para verificar que los cambios en la base de datos y de las variables operacionales sean los esperados.

- Paso 5. *Generar el reporte de requisitos*. Consiste de un informe en el que se detalla el primer acercamiento a la descripción de requisitos que se extrae desde las proposiciones, el tipo al que corresponde (funcional o no-funcional), y sus relaciones. Es una tabla que sirve de complemento a la plantilla y que juntas son el valor agregado de REDOC para la siguiente etapa

de la Ingeniería de Requisitos, y para el posterior documento de especificación. Conviene aclarar que debido a la volatilidad de los requisitos y al constante cambio en las necesidades de los clientes, todo este proceso de documentación se puede actualizar dinámicamente, porque sin importar que la acción sea *bottom-up* o *up-bottom* se reflejará fácilmente en los demás pasos.

3.2 Aplicación

La red de cajeros automáticos ATM es uno de los tres pilares de la infraestructura bancaria de cualquier país. Es común el uso de un cajero automático para facilitarles a los usuarios del sistema financiero hacer transacciones de manera conveniente, y es un elemento para promover la inclusión financiera. Mínimamente cuenta con una pantalla, un teclado como interfaz principal, un lector de tarjetas, y una impresora para imprimir los recibos de las transacciones. Antes de realizar operaciones a través del ATM el cliente tiene que iniciar sesión en el sistema. Para hacerlo desliza o introduce la tarjeta para validar sus datos. Si la tarjeta no es válida, se devuelve al cliente; y si la clave es incorrecta tres veces seguidas el ATM cancela la transacción y niega el acceso al sistema. Entre otras, las operaciones apoyadas por el ATM son: depósito, retiro, consulta de saldo, y cambio de clave, y cada una se registra para su posterior verificación y validación. En esta investigación y para aplicar el modelo REDOC, se toma como caso de estudio el *módulo retiro*.

- *Elaborar el diagrama de procesos*. En la Figura 1 se describe la secuencia de actividades incluidas en el módulo retiro de un ATM. Este diagrama representa las relaciones entre los componentes de todo el sistema, pero se debe reconocer que el mismo se encuentra inmerso en otro más complejo, que involucra a la entidad bancaria y sus relaciones con otros sistemas. La práctica de una representación visual les facilita a las partes interesadas comprender de mejor forma la descripción abstracta del problema e interpretar los requisitos que debe satisfacer la solución.

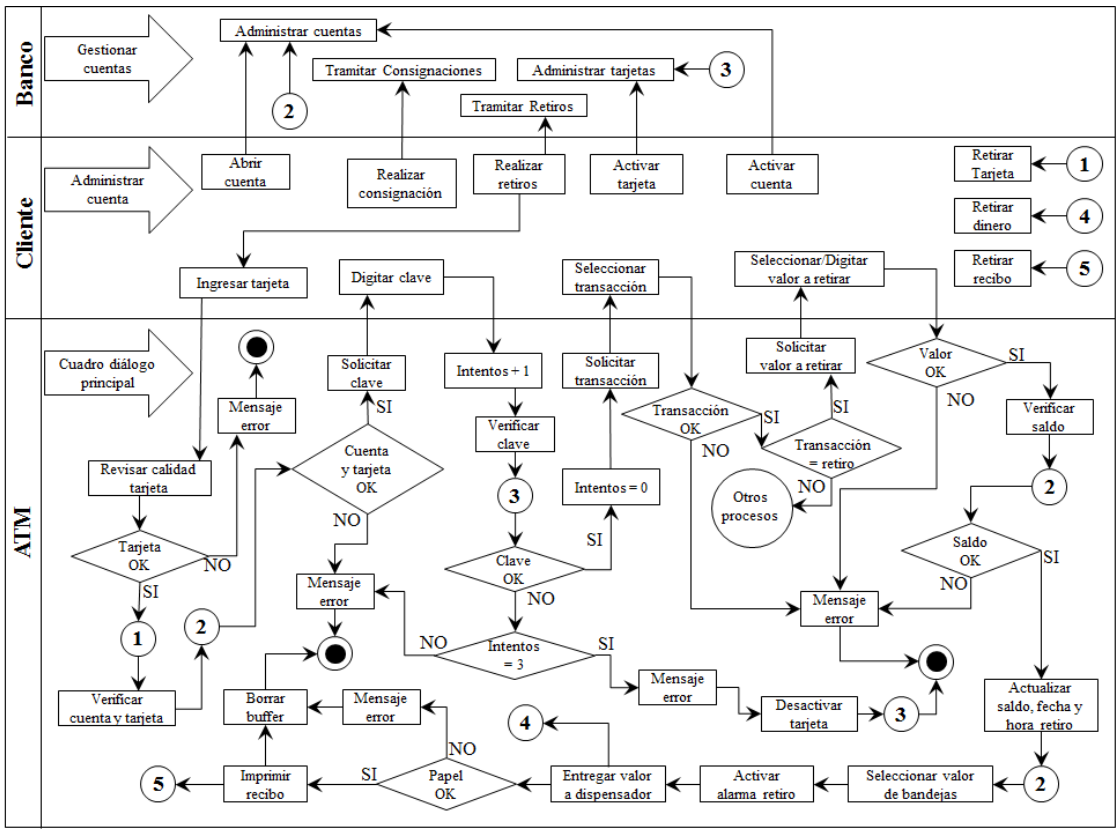

Figura 1. Diagrama de procesos del módulo Retiro en el ATM

- *Aplicar cálculo proposicional*. Luego de utilizar alguna de las técnicas de elicitación, de observar a los usuarios en el contexto del problema, y de escuchar sus descripciones en lenguaje natural, se construye el diagrama de procesos, desde el cual se extrae las proposiciones que se muestra en la Tabla 1.

Tabla 1. Necesidades como proposiciones

Caminos	Proposiciones
Principal	1. El ATM despliega cuadro de diálogo principal
	2. El CLIENTE introduce la tarjeta en el ATM
	3. El ATM verifica calidad de la tarjeta
	4. El CLIENTE retira la tarjeta
	5. El ATM verifica estado de cuenta y tarjeta
	6. El ATM despliega cuadro de diálogo digitar clave
	7. El CLIENTE digita clave
	8. El ATM incrementa número de intentos en 1
	9. El ATM verifica clave
	10. El ATM despliega cuadro de diálogo *seleccionar transacción*
	11. El CLIENTE selecciona transacción
	12. El ATM verifica transacción
	13. EL ATM despliega cuadro de diálogo *seleccionar/digitar valor retiro*
	14. El CLIENTE selecciona/digita valor retiro
	15. El ATM verifica valor retiro
	16. El ATM verifica saldo de cuenta
	17. El ATM actualiza el nuevo saldo
	18. El ATM actualiza fecha y hora retiro
	19. El ATM selecciona de las bandejas los billetes de valor retiro
	20. El ATM activa la alarma retirar dinero
	21. El CLIENTE retira el dinero
	22. EL ATM desactiva la alarma retirar dinero
	23. El ATM verifica la existencia de papel
	24. El ATM imprime recibo transacción
	25. El CLIENTE retira recibo transacción
	26. El ATM iguala las variables al valor definido
	27. El ATM despliega cuadro de diálogo principal
Alternativo 1	1.1 La tarjeta está en mal estado
	1.2 El ATM despliega mensaje de error
	1.3 El ATM finaliza proceso
	1.4 El ATM despliega cuadro de diálogo principal
Alternativo 2	2.1 La cuenta o tarjeta está desactivada
	2.2 El ATM despliega mensaje de error
	2.3 El ATM finaliza proceso
	2.4 El ATM despliega cuadro de diálogo principal
Alternativo 3	3.1 La clave digitada es incorrecta y número de intentos es menor a 3
	3.2 El ATM despliega mensaje de error
	3.3 El ATM finaliza proceso
	3.4 El ATM despliega cuadro de diálogo principal
Alternativo 4	4.1 La clave digitada es incorrecta y el número de intentos es igual a 3
	4.2 El ATM despliega mensaje de error
	4.3 El ATM desactiva la tarjeta
	4.4 El ATM finaliza proceso
	4.5 El ATM despliega cuadro de diálogo principal
Alternativo 5	5.1 La transacción seleccionada es incorrecta
	5.2 El ATM despliega mensaje de error
	5.3 El ATM finaliza proceso
	5.4 El ATM iguala número de intentos a 0
	5.5 El ATM despliega cuadro de diálogo principal

Alternativo 6	6.1 El valor a retirar es incorrecto
	6.2 El ATM despliega mensaje de error
	6.3 El ATM finaliza interacción *Realizar Retiros*
	6.4 El ATM iguala número de intentos a 0
	6.5 El ATM despliega cuadro de diálogo principal
Alternativo 7	7.1 El valor digitado para retirar es mayor al saldo de la cuenta
	7.2 El ATM despliega mensaje de error
	7.3 El ATM finaliza interacción *Realizar Retiros*
	7.4 El ATM iguala número de intentos a 0
	7.5 El ATM despliega cuadro de diálogo principal
Alternativo 8	8.1 El ATM no tiene papel
	8.2 El ATM despliega mensaje de error
	8.3 El ATM finaliza interacción *Realizar Retiros*
	8.4 El ATM iguala número de intentos a 0
	8.5 El ATM iguala las variables al valor definido
	8.6 El ATM despliega cuadro de diálogo de bienvenida

Debido a que el trabajo con las partes interesadas es iterativo, estas proposiciones se evalúan permanentemente hasta resumirlas en las necesarias para solucionar el problema, y que contengan las acciones que responden a las necesidades de las partes.

En la Tabla 2 se detalla las proposiciones resultantes, luego de las iteraciones.

Tabla 2. Proposiciones iteradas

Prioridad	Proposición resultante
1	El ATM despliega cuadro de diálogo principal
2	El ATM verifica calidad de la tarjeta
3	El ATM verifica estado de cuenta y tarjeta
4	El ATM despliega cuadro de diálogo digitar clave
5	El ATM incrementa número de intentos en 1
6	El ATM verifica clave
7	El ATM despliega cuadro de diálogo seleccionar transacción
8	El ATM verifica transacción
9	EL ATM despliega cuadro de diálogo seleccionar/digitar valor retiro
10	El ATM verifica valor retiro
11	El ATM verifica saldo de cuenta
12	El ATM actualiza el nuevo saldo
13	El ATM actualiza fecha y hora retiro
14	El ATM selecciona de las bandejas los billetes de valor retiro
15	El ATM activa la alarma retirar dinero
16	El ATM desactiva la alarma retirar dinero
17	El ATM verifica la existencia de papel
18	El ATM imprime recibo transacción
19	El ATM iguala las variables al valor definido
20	El ATM despliega cuadro de diálogo principal
2.1	El ATM despliega mensaje de error y finaliza proceso
6.1	El ATM desactiva la tarjeta
8.1	El ATM iguala número de intentos a 0
10.1	El ATM finaliza interacción Realizar Retiros
17.1	El ATM despliega mensaje de error

- *Elaborar el diagrama de caminos.* El primer acercamiento visual al modelamiento del problema, el diagrama de procesos, permite comprender el contexto del problema, mientras que el diagrama de caminos modela la posible solución. Para el caso de estudio y de acuerdo con las proposiciones iteradas, el diagrama de caminos resultantes se muestra en la Figura 2.

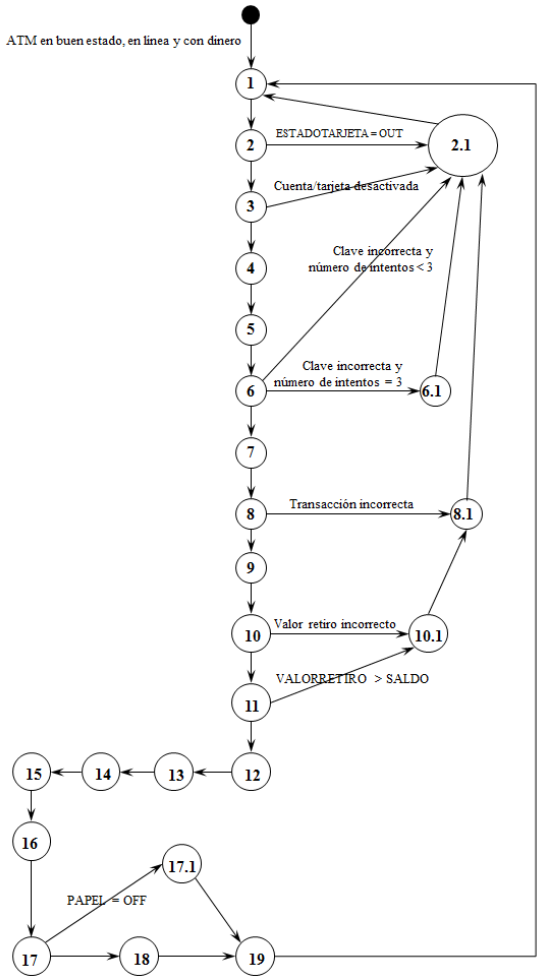

Figura 2. Diagrama de caminos resultante

- *Completar la plantilla de elicitación*. Para documentar el proceso de comunicación actor-sistema-actor, REDOC propone una plantilla para capturar la documentación del proceso de la elicitación de requisitos, tal como se muestra en la Tabla 3.

Tabla 3. Plantilla para documentar las interacciones

Código	001-1	**Nombre**	Realizar retiros	**Descripción**	El cliente retira dinero de su cuenta a través de un ATM	
Actores	Banco, Cliente, ATM		**Pre-condiciones**	ATM en buen estado, ATM en línea, ATM con dinero		
Dependencias anteriores	Abrir cuenta, cuenta y tarjeta activas, realizar consignación					
Camino Principal	**Acción Actor**		**Regla del negocio**		**Acción/respuesta Sistema**	**Cambio de estado**
	1. Ingresar tarjeta		1. CALIDADTARJETA = SI		1. Verificar calidad tarjeta electrónicamente	
	2. Retirar tarjeta		2. Cuenta ^ tarjeta activas		2. Verificar ESTADOCUENTA y TARJETA: BD	
					3. Cuadro de diálogo digitar CLAVE	
	3. Digitar CLAVE		3. CLAVE = 9999, INTENTOS < 3		4. Verificar CLAVE: BD	INTENTOS + = 1
					5. Cuadro de diálogo TRANSACCIÓN	INTENTOS = 0
	4. Seleccionar TRANSACCIÓN		4. TRANSACCIÓN selección opciones		6. Cuadro de diálogo VALORRETIRO	
	5. Seleccionar/digitar VALORRETIRO		5. VALORRETIRO (opción V múltiplo 10K)		7. Verificar VALORRETIRO	
			6. VALORRETIRO < SALDO		8. Verificar VALORRETIRO < SALDO: BD	
			6. SALDO = SALDO - VALORRETIRO		9. Actualizar SALDO: BD	SALDO = SALDO - VALORRETIRO

		7. FECHA = Date() ∧ TIME = Time()	10. Actualizar FECHA y HORA de retiro: BD	FECHA = Date() TIME = Time()
			11. Seleccionar VALORRETIRO en billetes	Dinero disponible - VALORRETIRO
			12. Llevar VALORRETIRO a dispensador	
			13. Activar alarma dispensador de retiro	Alarma activada
	6. Retirar dinero		14. Verificar PAPEL	
			14. Imprimir recibo transacción	Alarma desactivada
	7. Retirar recibo		15. Borrar buffer	Valores a 0 Variables en blanco
Caminos Alternativos		1. CALIDADTARJETA = NO	1.1 Mensaje Error; FIN proceso	
		2. Cuenta ∨ Tarjeta desactivada	2.1 Mensaje error; FIN proceso	
		3. Clave incorrecta ∧ INTENTOS < 3	3.1 Mensaje error; FIN proceso	
		4. Clave incorrecta ∧ INTENTOS = 3	4.1 Mensaje error; FIN proceso	TARJETA = OFF
		5. TRANSACCIÓN incorrecta	5.1 Mensaje error; FIN proceso	INTENTOS = 0
		6. VALORRETIRO incorrecto	6.1 Mensaje error FIN interacción Retirar Dinero	INTENTOS = 0
		7. VALORRETIRO > SALDO	7.1 Mensaje error FIN interacción Retirar Dinero	INTENTOS = 0
		8. PAPEL = OFF	8.1 Mensaje error FIN proceso Retirar Dinero	Valores a 0 Variables a blanco
Dependencias posteriores	Cambiar tarjeta; Reactivar cuenta; Reactivar tarjeta			
Post-condiciones	SALDO de cuenta actualizado; FECHA y HORA actualizadas; Valores a 0; Variables en blanco			
Observaciones				

- *Acción actor*. En esta columna se detalla la acción que el actor realiza como parte de su interacción con el sistema. Para documentarla se toma el principio del cálculo proposicional de la matemática discreta para anotar las acciones como proposiciones, lo que permite representar los requisitos sin ambigüedades.

- *Reglas del negocio*. Para documentar esta columna se utiliza principios de lógica y abstracción, el diseño por contrato y los métodos formales para convertir las proposiciones en fórmulas matemáticas. De esta manera es posible verificar las operaciones y si los valores resultantes son los esperados para los casos de prueba. Los caminos del proceso principal y los alternos, descritos en el diagrama de caminos, ocurren de acuerdo a las condiciones documentadas y toman el camino apropiado en cada circunstancia, además, deben respetar las reglas de negocio establecidas por el cliente y los usuarios.

- *Acción-respuesta sistema***.** En esta columna se utiliza el concepto de escenarios, tomado como buena práctica desde la revisión de literatura, y se describe las acciones-respuesta que el sistema ejecuta como procesos internos o como respuesta a una acción previa del actor. Los escenarios permiten entender la aplicación y su funcionalidad, y describen cada situación específica de la aplicación, centrando la atención en su comportamiento y asegurando el entendimiento y la colaboración entre las partes involucradas.

- *Cambios de Estado*. Conocer los cambios de estado es necesario porque REDOC asume las pruebas y la gestión de calidad como procesos paralelos a todo el ciclo de vida del producto.

Con el objetivo de poder verificar que sean los esperados para el sistema, en esta columna se documenta los cambios que ocurren en la base de datos y en las variables operacionales. Toda la información se expresa mediante fórmulas matemáticas para estructurar el plan de pruebas necesario, y respetando las reglas del negocio. Desde la acción actor y la acción-respuesta del sistema es posible conocer los posibles cambios de estado en la información que registra el sistema, que luego será validada cuando el proceso, que origina o modifica esa acción, recorra la ruta trazada en el diagrama de caminos.

- *Generar el reporte de requisitos*. Es conveniente aclarar que este reporte no reemplaza en ninguna medida al documento de especificación, solo representa el resultado del proceso de documentación de la elicitación. En la tabla 4 se presenta el formato de este reporte.

Tabla 4. Reporte de requisitos

ID	Requisito-proposición	Tipo Requisito	Relaciones	Justificación
1	El ATM verifica electrónicamente la calidad de la tarjeta	No-funcional	2	Es una rutina externa al sistema-solución
2	CALIDADTARJETA = SI	Funcional	1	
3	CALIDADTARJETA = NO	Funcional	1-4-5	
4	El sistema genera un Mensaje Error	Funcional	1-3	
5	La Cuenta ^ la tarjeta están activas	Funcional	6-7-8-9	
6	El banco verifica ESTADOCUENTA y TARJETA	No-funcional	1-7	Es una rutina externa al sistema-solución
7	La Cuenta v la Tarjeta están desactivadas	No-funcional	6-8	Es una rutina externa al sistema-solución
8	El sistema genera un Mensaje error	Funcional	7-9	
9	El sistema muestra Cuadro de diálogo digitar CLAVE	Funcional	1-6-10	
10	La CLAVE = 9999 ^ INTENTOS < 3	Funcional	9-11	
11	El banco Verifica CLAVE	No-funcional	10	Es una rutina externa al sistema-solución
12	El sistema incrementa INTENTOS + = 1	Funcional	10	
13	La Clave es incorrecta ^ INTENTOS < 3	Funcional	11	
14	El sistema genera Mensaje error	Funcional	13	
15	La Clave es incorrecta ^ INTENTOS = 3	Funcional	10-11-16	
16	El sistema genera Mensaje error	Funcional	15	
17	El banco hace TARJETA = OFF	No-funcional	15	Es una rutina externa al sistema-solución
18	El sistema muestra Cuadro de diálogo TRANSACCIÓN	Funcional	10	
19	El sistema hace INTENTOS = 0	Funcional	18-20	
20	El sistema muestra Cuadro de diálogo VALORRETIRO	Funcional	10-18-21	
21	El sistema recibe VALORRETIRO	Funcional	20	
22	El sistema Verifica VALORRETIRO	Funcional	21	
23	El VALORRETIRO es incorrecto	Funcional	22-24	
24	El sistema muestra Mensaje error	Funcional	23	
25	El banco Verifica VALORRETIRO < SALDO	No-funcional	21	Es una rutina externa al sistema-solución
26	El VALORRETIRO > SALDO	Funcional	25-27	
27	El sistema muestra Mensaje error	Funcional	26	
28	El banco hace SALDO = SALDO – VALORRETIRO	No-funcional	22-29	Es una rutina externa al sistema-solución
29	El banco hace FECHA = Date() ^ TIME = Time()	No-funcional	22-28	Es una rutina externa al sistema-solución
30	El sistema Lleva VALORRETIRO a dispensador	Funcional	29-31	
31	El sistema activa alarma dispensador de retiro	No-funcional	30	No todos los ATM utilizan esta función
32	El sistema Verifica PAPEL	No-funcional	31-33	No todos los ATM utilizan esta función
33	El sistema Imprime recibo transacción	No-funcional	32	No todos los ATM utilizan esta función
34	El sistema lleva los Valores a 0 ^ coloca Variables en blanco	Funcional	28-33-35	
35	El sistema Borra buffer	Funcional	34	

El lenguaje semi-formal con el que se expresa los requisitos en este reporte le permite al diseñador comprender el contexto sobre el cual debe trabajar, ya que elimina muchas de las ambigüedades del lenguaje natural, con el que suelen expresar los clientes sus necesidades.

3.3 Validación

Para llevar a cabo esta actividad se tomaron cinco de los modelos de la muestra de la revisión sistemática de la literatura de Serna y Suaza, con el objetivo de comparar y analizar los resultados en este mismo caso de estudio, y de acuerdo con una serie de variables, seleccionadas de otros trabajos en los que se hacen comparaciones similares [19, 20], que se adaptaron al objetivo de esta investigación. En la Tabla 5 se muestra las variables que se utilizaron para validar los resultados de aplicación de los diferentes modelos vs el modelo semi-formal propuesto.

Tabla 5. Variables de validación

Variable	Detalle
Modelado gráfico	Nivel de utilización de la representación gráfica para describir el proceso
Legibilidad de los requisitos	Los requisitos documentados se describen sin ambigüedades
Resolución de ambigüedades	Las ambigüedades se resuelven antes de la documentación
Capacidad para facilitar la comunicación	El lenguaje y los gráficos son claros, concisos, y precisos
Tipo de relación entre los requisitos	Los requisitos documentados tienen relación y continuidad
Representación por tipos de requisitos	Hace una clasificación de los requisitos de acuerdo a su tipo
Definición semántica	Detalla y explica todos los símbolos y palabras que puedan confundir
Trazabilidad de Requisitos	Permite conocer el histórico, la ubicación, y la trayectoria de cada requisito
Continuidad de etapa	Permite avanzar de un paso a otro de forma trasparente
Aporte a la especificación	Genera documentación de la elicitación aplicable y útil para la especificación

El proceso consistió en verificar la eficiencia y eficacia de cada modelo para documentar la elicitación, en comparación con los resultados obtenidos en modelo REDOC. Debido a que en la revisión de la literatura se encontraron pocas propuestas en este sentido, se seleccionaron las cinco consideradas como las más completas por sus pasos y productos:

- D1: Decision Tables in Software Engineering [21].
- D2: Object-Oriented Software Engineering: A Use Case Driven Approach [22].
- D3: Viewpoints for Requirements Elicitation: A Practical Approach [23].
- D4: Writing a Software Requirements document [24].
- D5: A Template for Requirement Elicitation Document of Software Product Lines [25].

4. ANÁLISIS DE RESULTADOS

D1. Las tablas de decisión no permiten el trabajo iterado entre ingenieros y clientes, porque las formas tabulares para organizar información son rígidas. Además, el modelo carece de un enfoque funcional, porque sus derivaciones son extensas y no involucran el uso de conceptos abstractos o teóricos, ni un lenguaje estructurado para documentar la elicitación. Se rescata la utilidad de los diagramas Chapin e HIPO para documentar los requisitos, los cuales se acercan al diagrama de procesos y al diagrama de caminos de REDOC, aunque requieren una filosofía de programación estructurada que desde hace tiempo no se utiliza masivamente. Como resultado, no fue posible identificar a tiempo las ambigüedades y la mayoría se camufla como requisitos no-funcionales. Además, la representación gráfica no es legible fácilmente, no es posible identificar claramente los tipos de requisitos, y no permite hacerles trazabilidad a los que aceptan las partes interesadas para entregarlos a la especificación.

D2. Al aplicar este modelo se genera un amplio número de diagramas, que de cierta forma y dependiendo del tamaño del sistema, dificulta la selección de los requisitos. Pero esta

representación gráfica es útil para personalizar los requisitos, y con el diagrama de casos de uso es posible modelar los funcionales, porque actúa como un puente entre los actores técnicos y el cliente. Las desventajas encontradas radican principalmente cuando se modela los requisitos no-funcionales, porque carece de una semántica formal definida para hacerlo, lo que no permite aclarar las diferencias entre las partes cuando se analiza las necesidades. Por otro lado, la documentación en las tablas no representa el dinamismo de las acciones y comunicaciones de las interacciones actor-sistema-actor, por lo que no es posible documentar el cambio de estado o las respuestas de las mismas.

D3. Identificar los problemas y malas interpretaciones de los requisitos, tan pronto como sea posible en la elicitación, es una de las mayores dificultades de este modelo, se acumula demasiada información debido a que cada punto de vista tiene una interpretación diferente, y al final se termina documentando información innecesaria. Aunque es posible agrupar la gestión de requisitos, la verificación de inconsistencias y la trazabilidad de los mismos, los puntos de vista reconocen múltiples perspectivas, lo que no proporciona un marco para el descubrimiento de conflictos en los requisitos propuestos; además, no permite la gestión de las inconsistencias. Otra dificultad es que es difícil definir la prioridad de los requisitos, por lo que la documentación no es suficiente para solucionarlo. El documento que se genera es extenso, porque atiende cada punto de vista de acuerdo con la herramienta PREview, lo que al final dificulta realizar cambios en los requisitos.

D4. El inconveniente que se encontró al aplicar este modelo es que no tiene en cuenta cómo documentar adecuadamente las definiciones de requisitos, porque el documento generado consiste solo de dos partes: una visión general y una descripción de la funcionalidad del sistema. Por otra parte, se debe incluir apéndices de acuerdo a sí se necesita más información de la que contiene el resto del documento o del material, que no encaja en ninguna otra parte. El documento generado es extenso y no ofrece claridad acerca de la comprensión de las necesidades, por lo que se debe re-leer continuamente hasta encontrar las diferencias de interpretación antes de especificar los requisitos.

D5. Este modelo utiliza plantillas para describir informalmente las líneas de productos software y para documentar su uso. El inconveniente encontrado es que la interacción resultante en los requisitos no se refleja en las tablas, por lo que cualquier modificación genera una nueva tabla de datos. Por otro lado, el uso exclusivo del lenguaje natural en cada una de ellas, no permite que se eliminen las ambigüedades en las negociaciones entre las partes, y al final lo que se obtiene es una especie de rompecabezas para entregar a la especificación.

En general, con ninguno de los modelos se pudo documentar la elicitación de tal forma que se pudiera considerar como base suficiente para construir el documento de la especificación. El principal inconveniente es que trabajan en lenguaje natural y no hacen una representación gráfica suficiente clara para comprender el problema, y modelar una posible solución. Por lo que las partes interesadas y los ingenieros necesitan más tiempo para analizar cada necesidad, antes de ubicarla como requisito. En la Tabla 6 se resume los resultados de la valoración obtenida.

La valoración de *Alto*, *Medio* y *Bajo* se utiliza para calificar el nivel de cumplimiento de las variables evaluadas. Por ejemplo, para el caso de la variable *modelado gráfico* se asigna A si utiliza la representación gráfica en todos los pasos para describir el proceso, M si la utiliza por lo menos en la mitad de ellos, y B si lo hace para menos de la mitad. Aunque esta valoración para otras variables puede ser subjetiva debido a la falta de experiencia en la utilización de las propuestas, se puede identificar una tendencia en la que sobresale las propuestas D2 y el modelo REDOC.

Tabla 6. Resumen de la validación de la aplicación de todos los modelos

Variable	D1	D2	D3	D4	D5	REDOC
Modelado gráfico	B	A	M	M	B	A
Legibilidad de los requisitos	M	M	M	M	M	M
Resolución de ambigüedades	B	M	B	B	B	A
Capacidad para facilitar la comunicación	B	M	B	M	B	M
Tipo de relación entre los requisitos	M	M	B	B	B	M
Representación por tipos de requisitos	B	M	B	B	B	A
Definición semántica	B	B	B	B	B	M
Trazabilidad de Requisitos	B	B	B	M	M	M
Continuidad de etapa	B	M	M	B	B	A
Aporte a la especificación	B	M	B	M	B	M

A: Alto, M: Medio, B: Bajo

En resumen y luego de analizar estos datos, se concluye que la eficiencia y eficacia para documentar la elicitación de requisitos del modelo REDOC es superior a la mayoría de los modelos propuestos en el mismo sentido. Esto valida la hipótesis de que una descripción semi-formal ayuda a eliminar la ambigüedad del lenguaje natural, lo que les permite a las partes y a los ingenieros llegar más fácilmente a acuerdos acerca de las necesidades que se deben caracterizar como requisitos, y a entregar un documento de elicitación desde el que se origina fácilmente la especificación.

5. CONCLUSIONES

Aunque en la comunidad se reconoce ampliamente la importancia de la Ingeniería de Requisitos y a la elicitación como una etapa importante de la misma, preocupa el hecho de que se presenta pocos modelos orientados a cómo documentar los requisitos. La importancia de una adecuada documentación en esta etapa radica en que permite una mayor comprensión de las necesidades del cliente, les ayuda a los ingenieros a percibir de mejor forma el problema y a modelar una solución que se refleje adecuadamente en la especificación. Además, brinda una mejor base para abordar las demás etapas de esta fase y de las otras del ciclo de vida, porque se puede consultar cada vez que no se comprenda las necesidades o cuando se deba atender modificaciones. Esto es necesario para que los proyectos software no excedan los tiempos ni los costos inicialmente establecidos [40].

El modelo semi-formal REDOC propuesto en esta investigación parte de incluir elementos formales a la descripción textual, como los de la plantilla para documentar las interacciones. Y como se observa en los resultados de la aplicación, esto presenta ventajas en relación con los modelos avaluados debido a que son estáticos y utilizan lenguaje natural, lo que genera diferencias considerables en la interpretación de los requisitos. Por su parte, al incluir características como almacenamiento, cambios de estado y reglas de negocio, REDOC representa de forma coherente, clara y completa la funcionalidad del proyecto, de tal manera que es escalable y adaptable al medio, de igual forma mejora la comunicación entre los actores involucrados para generar el reporte de requisitos.

Después de haber aplicado el modelo y del posterior análisis de los resultados, las posibles líneas de trabajo futuras para abordar de forma conjunta el proceso de documentar la etapa de elicitación de requisitos son:

- Incrementar la investigación en métodos formales en la Ingeniería de Requisitos, porque en otras áreas de la Ingeniería del Software se aplican desde antes, y aunque tienen un amplio

recorrido y su utilidad y eficiencia en desarrollos críticos están demostradas, todavía falta más trabajo para que la mayoría de ingenieros los conozcan y apliquen.

- La automatización de las pruebas debe ser incluida en un nuevo modelo para gestionar requisitos. Los modelos actuales de pruebas de software disponen de metodologías que sugieren adquirir distintos niveles de madurez en los procesos, sin embargo, no especifican detalles, lo cual conduce a que el desarrollo de los mismos sea complejo.

- Capacitar mejor a los estudiantes para trabajar con la matematización de la Ingeniería del Software, de tal forma que se pueda lograr que la labor y las actividades que desarrolla un ingeniero de Software sean verdadera ingeniería, y que al usuario final se le entregue productos software confiables y de acuerdo con las necesidades establecidas.

- Es necesario validar el modelo REDOC en casos reales de la industria, para de esta forma verificar la consistencia lógica en todos sus componentes, y para encontrar las falencias y complicaciones en surgen en un desarrollo real.

- Para determinar la eficiencia y eficacia de las proposiciones que entrega la etapa de elicitación, es necesario experimentar con procesos de inclusión de los resultados del modelo en las técnicas de especificación y en el documento de especificación.

REFERENCIAS

[1] Mustelier D. y Viera Y. (2013). Variables that define the complexity of the software functional requirements. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 3(2), 38-42.

[2] Serna E. (2010). Formal Methods and Software Engineering. Revista Virtual Universidad Católica del Norte 30, 158-164.

[3] Bolstad M. (2004). Design by Contract: A simple technique for improving the quality of software. En 31st Annual International Symposium on Computer Architecture. Williamsburg, USA.

[4] Serna E. (2018). Metodología de investigación aplicada. En E. Serna (Ed.), Ingeniería: Realidad de una Disciplina (pp. 6-33). Editorial Instituto Antioqueño de Investigación.

[5] Serna E. (2012). Analysis and Selection to Requirements Elicitation Techniques. En 7th Colombian Computing Congress. Medellín, Colombia.

[6] Davis A. et al. (2004). Great Software Debates. John Wiley.

[7] PMI. (2009). Guía de los fundamentos para la dirección de proyectos. Guía del PMBOK®. Project Management Institute.

[8] Software Engineering Institute. Recuperado: http://www.sei.cmu.edu/cmmi/

[9] IEEE. (2014). Guide to the Software Engineering Body of Knowledge SWEBOK®. IEEE Computer Society.

[10] Burgess C. (1995). The Role of Formal Methods in Software Engineering Education and Industry. En 4th Software Quality Conference. Dondee, Scotland.

[11] Soares M. y Sousa D. (2012). Analysis of Techniques for Documenting User Requirements. Lecture Notes in Computer Science 7336, 16-28.

[12] Smith C. y Williams L. (2003). Best Practices for Software Performance Engineering. En 29th International conference of Computer Measurement Group. Dallas, USA.

[13] Lloyd J. (1994). Practical advantages of declarative programming. En Joint Conference on Declarative Programming. Valencia, España.

[14] Mitchell R. y McKim J. (2001). Design by Contract by Example. Addison Wesley.

[15] Burgess C. (1995). The Role of Formal Methods in Software Engineering Education and Industry. En 4th Software Quality Conference. Dondee, Scotland

[16] Soares M. y Sousa D. (2012). Analysis of Techniques for Documenting User Requirements. Lecture Notes in Computer Science 7336, 16-28

[17] Smith C. y Williams L. (2003). Best Practices for Software Performance Engineering. En 29th International conference of Computer Measurement Group. Dallas, USA.
[18] Lloyd J. (1994). Practical advantages of declarative programming. En Joint Conference on Declarative Programming. Valencia, España.
[19] Soares M. y Sousa, D. (2012). Analysis of Techniques for Documenting User Requirements. Lecture Notes in Computer Science 7336, 16-28
[20] Smith C. y Williams L. (2003). Best Practices for Software Performance Engineering. En 29th International conference of Computer Measurement Group. Dallas, USA.
[21] Hurley R. (1982). Decision Tables in Software Engineering. Van Nostrand Reinhold.
[22] Jacobson I. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley.
[23] Sommerville I. et al. (1998). Viewpoints for Requirements Elicitation: A Practical Approach. En 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice. Colorado Springs, USA.
[24] Berezin T. (1999). Writing a Software Requirements document. Recuperado: https://home.adelphi.edu/~siegfried/cs480/ReqsDoc.pdf
[25] Gallina B. et al. (2007). A Template for Requirement Elicitation Document of Software Product Lines. Technical Report. TR-LASSY-06-08. University of Luxembourg.

CAPÍTULO XIX
Marco de trabajo para elicitar requisitos multidimensionales[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

Partiendo del hecho de que los requisitos no tienen una única y aislada fuente de origen, es decir, no son unidimensionales como se asume en la elicitación tradicional, en este capítulo se propone elicitarlos desde su multidimensionalidad. Se presenta una revisión de la literatura acerca de los trabajos que relacionan a la multidimensionalidad con las necesidades de las partes interesadas, una visión de los requisitos multidimensionales y se describe el proceso para llevar a cabo su elicitación. Se encontró que estas propuestas relacionan la multidimensionalidad en la elicitación, pero no la tienen en cuenta para construir una especificación de requisitos representativa. Es evidente que no existe unanimidad acerca de las dimensiones desde las que se originan los requisitos y tampoco se gestionan para elicitarlos. Por eso y debido a que se acepta su existencia, se unifica los criterios descubiertos y se propone un marco de trabajo para elicitar requisitos multidimensionales.

[1] Publicado en inglés en la Revista Ingeniería y Universidad 22(2), 1-13. 2018.

INTRODUCCIÓN

En la Ingeniería del Software el término *requisitos* se refiere al conjunto de necesidades que debe satisfacer el producto en desarrollo, que en términos generales provienen de las personas, la arquitectura tecnológica, otros sistemas, el contexto político-administrativo nacional e internacional, la responsabilidad social y ético-ecológica, entre otras. Aunque la mayoría de modelos y metodologías tienden a tratarlas como *fuentes de origen* de los requisitos, en realidad deberían considerarse como *dimensiones*, porque son estructuras en las que se categorizan hechos, acciones y normas con el propósito de servir de oráculos para los analistas.

En el caso del desarrollo de software el concepto dimensión no se debe confundir con el aplicado en física, matemáticas o filosofía, sino como una especie de enciclopedia de datos que contiene información relacionada con el tipo de problema a solucionar. Es decir, es una colección de información de referencia acerca de diversos eventos medibles, tales como los requisitos del software. Cada dimensión categoriza y describe hechos, acciones y normas acerca de los datos que contiene, de tal manera que pueden ser consultadas iterativamente para explorar, discutir, aclarar, definir y acordar lo que el producto final debe satisfacer.

Un asunto importante a tener en cuenta en la elicitación es que no se puede suponer que el conjunto de requisitos es definitivo y completo, porque, de hecho, cambia a medida que progresa la solución, durante el desarrollo del proyecto, o como parte de la evolución normal del contexto en el que se inscribe el problema [1].

Para incrementar la posibilidad de acercarse a lo esperado, y debido a las dificultades de que un proyecto software se ajuste al presupuesto y a los tiempos de entrega a la vez que satisface completamente los requisitos, es necesario que las partes interesadas trabajen juntas y que analicen sus necesidades desde cada dimensión involucrada. De esta manera pueden lograr una visión compartida desde el análisis de la información consultada, porque discuten e interpretan multidimensionalmente las perspectivas y necesidades de cada uno.

En la literatura sobre elicitación se da a entender que la labor del analista en esta etapa es como la de un agricultor que camina a través de un huerto buscando y recogiendo los frutos maduros. En la práctica este proceso no es tan simple, debido a que el analista tiene que desarrollar las habilidades y tenacidad de un detective, porque descubrir y comprender requisitos es una tarea compleja.

No se trata de buscarlos simplemente en fuentes de origen únicas, porque sería como intentar encontrar frutos diversos en un único árbol. Por el contrario, hay que buscarlos e identificarlos en dimensiones, porque están inmersos en documentos, artefactos, experiencias, imágenes, interpretaciones y pensamientos, que generalmente no son su origen directo, y porque tienen interrelaciones y dependencias complejas con otras dimensiones y disciplinas.

En este Capítulo se afirma que los requisitos no se deben elicitar buscándolos solamente en una fuente de origen, sino que primero hay que identificar las dimensiones en las que pueden existir, para luego definir y discutir su pertinencia para el proyecto. En cada dimensión pueden tener relaciones y dependencias múltiples, además de fuentes directas e indirectas que en muchas ocasiones no son explícitas. Resolver estas dificultades es una labor detectivesca y el analista debe conocer esas dimensiones para poder identificar y documentar los requisitos. Además, debe trabajar con un equipo transdisciplinar para descubrir la información tácita y explícita requerida, que le permita analizar, estructurar y proponer una solución al problema.

1. MÉTODO

De acuerdo con Kitchenham [2], una revisión de la literatura se desarrolla mediante tres fases sprimarias: 1) planear 2) realizar, y 3) documentar. Además, estas fases se deben ejecutar mediante una serie de pasos, los cuales se describen en la Tabla 1.

Tabla 1. Metodología de investigación

Pasos	Descripción
Preguntas de investigación	Qué enfoques se propone en la literatura para elicitar requisitos multidimensionales
Proceso de búsqueda	Este propósito se diseña un plan de búsqueda en las bases de datos: ACM, IEEE, Science Direct, Springer y Wiley. Los parámetros de búsqueda incluyen palabras como {multidimensionality + requirements + elicit + management + approaches + models + methodologies + frameworks}, que deben aparecer en el título, el resumen, o en las palabras clave del documento.
Criterios de inclusión-exclusión	El principal criterio de inclusión es la relevancia del trabajo para responder la pregunta de investigación. Sin embargo, también se consideraron criterios como: el trabajo es una investigación explicita; presenta una descripción teórica; describe una aplicación práctica; discute un caso de estudio; propone un modelo, marco de trabajo o metodología; tiene un buen número de citaciones. Inicialmente, los trabajos que no cumplen alguno de estos criterios son descartados.
Evaluación a la calidad	Para determinar la calidad de los trabajos se consideraron criterios como: formalidad y pertinencia del medio de divulgación; la autoridad de los autores; calidad de los resultados y los datos fuente; grado de aceptación (citaciones); proceso de investigación aplicado; coherencia entre resultados y conclusiones; comentarios y reconocimiento de la comunidad. A cada uno se le asignó un valor para determinar la calidad.
Recopilación de datos	En una matriz se recopiló: 1) tipo (artículo, libro, capítulo libro, presentación en evento; otro), 2) título, 3) autor, 4) contribución (teórica, práctica, estudio de caso, modelo, metodología. Se encontraron 23 documentos.
Análisis de datos	Al aplicar los criterios de inclusión-exclusión y la evaluación a la calidad se extrajeron cinco trabajos. Luego se analizó el contenido para determinar su relevancia para responder la pregunta de investigación, con lo que se extrajeron otros tres documentos.

2. RESULTADOS Y ANÁLISIS

Algunos autores relacionan en sus trabajos la dimensionalidad de los requisitos, tales como Klaus Pohl [3], que propone un marco en el que la especificación es un proceso que se origina en ideas vagas en lenguaje natural. Su propuesta se caracteriza por enmarcar el proceso de la Ingeniería de Requisitos en tres dimensiones: 1) *Especificación*, donde ubica los métodos utilizados y los puntos de vista para comprender las necesidades del sistema; 2) *Representación*, donde organiza los requisitos utilizando alguna forma de notación; y 3) *Conformidad*, donde aborda cómo llegar a un acuerdo común sobre los requisitos y los objetivos fundamentales del sistema.

Para Durán [4] el término requisito recibe una amplia cantidad de calificativos que reflejan aspectos dimensionales, pero que generalmente se consideran de forma aislada. El autor identifica tres dimensiones desde las que se originan los requisitos: 1) *Ámbito*, componente en el que se debe cumplir; 2) *Característica*, rasgos del sistema que se desean en la especificación; y 3) *Audiencia*, actores específicos que los puedan comprender. Para Sawyer y Kotonya [5] existe un fuerte solapamiento entre la clasificación y los atributos de los requisitos, debido a que su origen se puede clasificar en dimensiones: 1) *Nivel de cumplimiento*, si es funcional o no-funcional; 2) *Nivel de surgimiento*, si son derivados o emergentes; 3) *Grado de exigencia*, si son del producto o del proceso; 4) *Prioridad*, si su importancia es alta, media o baja; 5) *Alcance*, si afectan al sistema o a un componente; y 6) *Volatilidad/estabilidad*, nivel de cambios que sufren en el desarrollo.

Silva et al. [6] afirman que la multidimensionalidad inherente a las aplicaciones presenta un nuevo reto para el análisis de los requisitos de software, porque la información en ellos es una cuestión de explotación e integración de datos que requiere conocimientos de alto nivel [7]. Para ellos, la elicitación es similar al manejo de un almacén de datos, en el que las dimensiones son

perspectivas individuales que determinan el nivel de detalle que se adoptará para su representación como requisitos. De acuerdo con Tuunanen [8] en la literatura existe un vacío acerca de cómo elicitar requisitos desde diferentes dimensiones. En su trabajo presenta un análisis desde dos de ellas: la comunicación y el alcance. La conclusión de este autor es que en la literatura no se explica cómo superar las barreras del alcance de los requisitos, ni cómo lograr que la comunicación sea fluida y en un lenguaje común y concertado en el grupo.

Para Nurmuliani y sus colegas [9] la gestión de los cambios en los requisitos es un problema multifacético, que se podría solucionar adoptando un enfoque multidimensional para la elicitación. Además, hay que tener en cuenta la naturaleza de los cambios, las características de los participantes y el contexto del diseño, porque son factores importantes para alcanzar una gestión adecuada. Por su parte, Deborah Coleman [10] afirma que los métodos de elicitación generan puntos de vista independientes de los requisitos, a pesar de que muchas de sus actividades, o dimensiones, se superponen en el contexto del proyecto. Concluye que elicitar requisitos de forma dimensional genera oportunidades de colaboración en el equipo, mejora la gestión del conocimiento en esta etapa y se beneficia la calidad del producto.

En este mismo sentido Moreira et al. [11] sostienen que un enfoque de Ingeniería de Requisitos efectivo debe conciliar la necesidad de lograr la separación de las interpretaciones con la de satisfacer los requisitos y las restricciones generales. Sin embargo, los métodos tradicionales no ofrecen ese apoyo para esta fase del ciclo de vida, porque la mayoría solamente tiene la visión de dos dimensiones: funcionales y no-funcionales. Su propuesta es que los requisitos se deben elicitar uniformemente desde todas las dimensiones origen, independientemente de su naturaleza. Annoni y sus colegas [12] sostienen que algunas investigaciones en Ingeniería de Requisitos se han centrado en el diseño de modelos conceptuales multidimensionales, pero que muy pocos se orientan a desarrollar modelos y herramientas para analizarlos. Llenar este vacío es el objetivo de su trabajo, para lo cual muestran cómo derivar sistemáticamente un esquema conceptual de requisitos multidimensionales.

Debido a la complejidad de los sistemas actuales es necesario explorar y analizar la variabilidad de los requisitos en un nivel más alto de abstracción, pero la dificultad para comprender su origen influye sobre las interpretaciones de los analistas [13]. Esto hace que sea difícil comprender y gestionar esa variabilidad, porque el proceso tradicional no tiene en cuenta las dimensiones particulares que definen el origen de los requisitos. Estos autores proponen abordar la variabilidad desde las dimensiones, para modelarla y generar representaciones susceptibles de razonamiento, a la vez que se aprovecha el nivel de variación elicitando términos específicos desde cada dimensión. Dufresne [13] modifica el Modelo de Kano et al. [15] acerca de la teoría de la calidad atractiva, y lo analiza para la Ingeniería de Requisitos. De acuerdo con él, el bajo nivel de aceptación del software se debe en parte a que los requisitos se elicitan unidimensionalmente, lo que no permite determinar sus interrelaciones y dependencias con y en otras dimensiones.

Pa y Zin [16] afirman que uno de los problemas de la elicitación de requisitos es la mala calidad de la comunicación entre los integrantes del equipo de trabajo. Para ellos esta práctica es más que expresarse, porque exige poner en común los valores de lenguaje, la experiencia y la cultura. Por eso recomiendan tener en cuenta las dimensiones desde las que se originan los requisitos (física, social, psicológica y temporal), para comprenderlos y derrumbar las barreras que puedan aparecer. Por su parte, Tran y Anvari [17] proponen un marco de elicitación de requisitos desde cinco dimensiones: gestión del cambio, características del usuario, conocimiento, proceso cognitivo y evaluación. Es una propuesta inspirada en diferentes técnicas adoptadas de diversas disciplinas, con el objetivo de ayudarles a los analistas. Para ellos, el hecho de que el origen de los

requisitos se analice desde una sola dimensión no permite una especificación estable ni duradera, porque se debe estar modificando cuando se descubre sus relaciones dimensionales.

Como se observa en esta revisión de la literatura los autores hacen relación a la multidimensionalidad de la elicitación o de la Ingeniería de Requisitos, pero sus aportes están más orientados al análisis y la representación que a la gestión para construir una especificación representativa de requisitos multidimensionales. Otra cosa que se descubre es que no existe unanimidad acerca de las dimensiones fuente de los requisitos, lo que está bien, porque cada problema a resolver presenta un contexto y un dominio particular. Lo que sí se deberían tener en cuenta es que existe dimensiones propias de la interacción humana que tienen repercusión en las soluciones software, tales como la socio-cultural, la político-institucional, la administrativa, la tecnológica y la disciplinar, que están inmersas en cualquier problema que se pueda resolver con estos productos, y en las cuales se originan sus requisitos.

2.1 La elicitación tradicional

Actualmente, la elicitación de requisitos se practica atendiendo procesos y metodologías generales que se adaptan a cada proyecto, pero que además toman y conjugan procesos y metodologías desde diferentes áreas del conocimiento. Es decir, esta actividad es una mezcla de lo que ha funcionado para otros, en combinación con lo que se cree que puede funcionar en la resolución de cada problema en particular. Pero esta manera de trabajar es difícil de articular en los contextos complejos de hoy, porque generalmente está orientada a encontrar las necesidades de las partes interesadas, para documentarlas y solucionarlas con el desarrollo de un producto, mientras que la complejidad del software exige una visión diferente para gestionar esos requisitos. En la práctica, generalmente la elicitación es un proceso de interacción en el que se aplica uno o varios métodos [18]:

- *Verbales*. Son técnicas de comunicación e interacción social, tales como entrevistas, talleres, grupos focales y lluvia de ideas, que utilizan los analistas para elicitar los requisitos explícitos [19]. Pero aplicarlas es una labor exigente, porque es tedioso organizar las reuniones y recolectar la información, además de reproducir y analizar su transcripción.

- *De observación*. Consisten de sesiones de observación de las actividades humanas, mediante las cuales se puede descubrir requisitos que difícilmente se pueden verbalizar, es decir, los implícitos [20]. Son adecuados cuando las partes no se expresan oralmente con facilidad, o cuando los analistas quieren comprender el contexto de utilización del sistema [21]. Pero debido a que normalmente son estudios longitudinales su aplicación requiere tiempo, algo de lo que no se dispone con frecuencia.

- *Analíticos.* Son diferentes formas de explorar la documentación o el conocimiento existentes alrededor de un problema, y para elicitar requisitos desde las deducciones de los analistas. Además, permiten recoger información del dominio del sistema, del flujo de trabajo y de las características del software. Aunque se consideran como no-vitales en la elicitación, pueden ser complementarios cuando se reutiliza información heredada o relacionada [22].

- *Sintéticos*. También se conocen como técnicas colaborativas, porque no combinan métodos individuales, sino que hacen una composición sistemática de los métodos anteriores para realizar un análisis como método individual. De esta manera aprovechan que, en la elicitación, todas las partes se comunican e interpretan de diversas maneras con el objetivo de encontrar un entendimiento común de los requisitos del producto [23]. Al combinar diferentes canales

de comunicación proporcionan modelos con los que es posible demostrar las características y las interacciones del sistema, pero es difícil encontrar analistas capacitados para aplicarlos.

La mayoría de analistas selecciona uno o varios de estos métodos por diferentes factores relacionados con el sistema a desarrollar: 1) *Nivel de abstracción*: porque abstraen el problema y establecen sus límites [24] mediante el conocimiento genérico del análisis del problema y el conocimiento específico de la descripción del producto. 2) *Las fuentes*: porque deben asegurarse de utilizarlas efectivamente en términos de los datos disponibles y del proceso para construir conocimiento. 3) *La comunicación*: porque en la interacción de los diferentes actores y los analistas existe barreras para intercambiar datos e información [25]. 4) *Familiaridad*: porque el nivel de certidumbre sobre el contexto y el dominio del problema refleja una situación problemática relativamente madura, y es fácil de comprender y estructurar, dando como resultado que la visión y el alcance del producto están bien expresados.

La selección y aplicación de estos métodos en la Ingeniería de Requisitos se basa en la suposición implícita de que, las partes interesadas, han conformado un equipo de trabajo cooperativo y sincero, que sus integrantes están dispuestos a compartir su conocimiento y que los analistas se preparan cuidadosamente para cada sesión de elicitación. La realidad es que pocas veces se logran estas situaciones, debido en parte a que los equipos se conforman alrededor de una disciplina dominante, y a que el conocimiento implícito que poseen los actores no se comparte adecuadamente. Esto hace que se deba experimentar hasta encontrar el método que ofrezca los mejores resultados en la búsqueda de requisitos, lo que genera reprocesos, retrasos en tiempo y sobrecostos.

En este sentido, Standish Group realizó varios estudios acerca de los éxitos, los desafíos y los fracasos en los proyectos software debidos a una elicitación inadecuada [26]. De acuerdo con estos informes, más de la mitad de los sobrecostos y fracasos se debe a errores en esta etapa de la Ingeniería de Requisitos, una afirmación que sustentan diversos estudios [27-31].

Los resultados al análisis de estos reportes dan como resultado una amplia lista de problemas identificados, que pone de manifiesto las falencias de la elicitación tradicional. Aunque los autores reportan los casos a la vez que su posible solución, la variedad y el alcance de las deficiencias son tan extensas que intentar solucionarlas se convierte en un esfuerzo complejo y difícil.

El alcance de presente trabajo no es analizar todos los problemas de la elicitación de requisitos, sino que se centra básicamente en la cuestión del manejo de las fuentes. Esta característica se presenta en todos los métodos actuales y aparece en la lista de los reportes de Standish Group y de otros analistas, los cuales la consideran como una causa de las deficiencias en esta etapa. Considerar que los requisitos provienen o se pueden buscar en alguna fuente determinada y única, es restringir la posibilidad de encontrar sus interrelaciones y dependencias dimensionales.

Además, en la elicitación se entremezcla aspectos humanos que impiden su ubicación en una simple fuente de origen; el lenguaje natural no siempre es adecuado en la expresión tecnológica, por lo que requiere *traducción*; las necesidades que debe resolver el sistema cambian en cualquier momento y desde cualquier dirección; los clientes describen sus necesidades como si se tratara de un universo único, cuando en realidad tienen relaciones que no se pueden identificar desde una única fuente; y debido a que la Ingeniería de Requisitos no es determinista, tampoco lo es la elicitación. Por eso es necesario innovar y aprovechar los aportes desde otras áreas del conocimiento para mejorar la elicitación y estructurar un marco de trabajo innovador.

3. MARCO DE TRABAJO PROPUESTO

La elicitación de requisitos es un proceso en el que el equipo de trabajo busca, revela, comprende y documenta las necesidades que debe satisfacer el sistema; las cuales, dependiendo de su impacto e incidencia en la solución, tradicionalmente se clasifican en funcionales y no-funcionales. Al elicitarlas como si su origen fuera de una fuente única se genera problemas de comprensión y de interpretación, porque la visión disciplinar con la que se conforma el equipo de trabajo tiende a inclinar la búsqueda en una sola dirección. Esto no permite descubrir y diagramar las múltiples interrelaciones que tienen los requisitos entre y desde esas fuentes. Por eso es que en la elicitación se debe encarar la búsqueda y comprensión de los requisitos con una visión multidimensional, de tal manera que el equipo pueda determinar su alcance y necesidad para la solución, antes de documentarlos en la especificación de requisitos.

En una sociedad software-dependiente, donde la complejidad y el tamaño de los problemas que se puede resolver con este producto tecnológico se incrementan cada vez, los métodos tradicionales para elicitar requisitos parecen ser de poca ayuda para los analistas. Este contexto exige un cambio en la práctica, pero se necesita una voluntad y una visión diferentes para emprenderlo. Esta falta de prospectiva en la industria y la academia acerca de la gestión y la administración del software degrada la fiabilidad de los procesos y del producto final. Hay que aceptar que la elicitación es más que un proceso de comunicación y observación, porque requiere que el equipo y las partes interesadas compartan características tales como lenguaje, experiencia, valores culturales, intereses y conocimiento.

La realidad es que esta etapa de la Ingeniería de Requisitos se lleva a cabo en un contexto particular y se desarrolla involucrando varias dimensiones: organizacional, física, social, psicológica, cultural, disciplinar, cognitiva y temporal, entre otras, que a su vez están inmersas en otras con mayor cobertura: globalización, jurídica, economía, transdisciplina, entre otras. En este contexto multidimensional los requisitos se interrelacionan y dependen de múltiples enlaces, que muchas veces no presentan un evento que los origine o finalice, o mucho menos que los contenga exclusivamente como fuente única. Aun así, la Ingeniería de Requisitos tradicional se ha centrado en un triple objetivo de capturar, analizar y gestionar la información de los requisitos para generar el documento de la especificación.

La Multidimensionalidad es un principio del Pensamiento Complejo [32] que brinda la posibilidad de acabar con determinismos y reduccionismos, y que para la elicitación pasa por la inclusión de dimensiones fuente que no agoten, sino más bien que aumenten, las posibilidades de encontrar los requisitos. Su objetivo es la comprensión del mundo, en este caso del problema, por lo que es imperativo llegar a una unidad del conocimiento sobre los requisitos del sistema. Esto implica ir más allá de las fuentes únicas y de las interpretaciones personales para lograr una verdadera integración. Además, la cadena dato-información-conocimiento no proviene de una fuente o dimensión única, sino de una variedad que se conjuga aleatoriamente, y al parecer sin un sentido práctico. Tener en cuenta y analizar cada requisito desde su(s) dimensión(es) origen les facilita a los analistas conformar una definición única del mismo e integrarlo en el documento de elicitación.

Por otro lado, en una elicitación multidimensional hay que tener en cuenta que los requisitos explícitos y los tácitos pueden existir en dimensiones internas o externas al problema. Esto se evidencia en las relaciones que las personas y las organizaciones crean, formal o informalmente, con sus proveedores, clientes y usuarios, y que involucran alguno de los aspectos relacionados con el problema. La labor detectivesca de los analistas es encontrar esas relaciones e investigarlas

hasta encontrar los requisitos. Pero deben ser precavidos, porque muchas se basan en la confianza e intentar romperla puede generar problemas de hermetismo. Por eso se recomienda adoptar prácticas que permitan descubrir lo que cada individuo sabe o conoce del problema, sin violentar sus relaciones o decisiones. Por ejemplo, solicitar un documento formal en lenguaje técnico, en el que describa los aspectos importantes de su trabajo, podría ofrecer los datos-información-conocimiento suficientes para descubrir los requisitos embebidos. En la Figura 1 se detalla el marco de trabajo para elicitar requisitos multidimensionales.

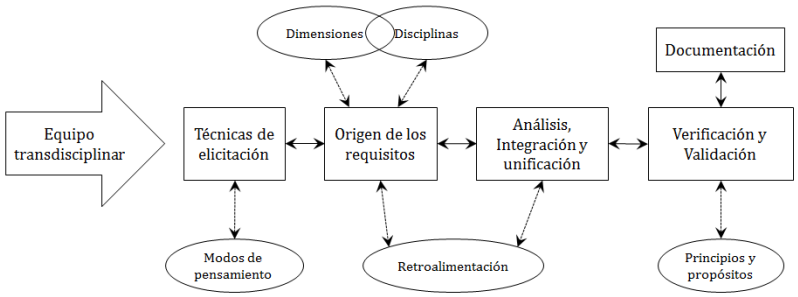

Figura 1. Marco de trabajo para elicitar requisitos multidimensionales

Este proceso se desarrolla conjuntamente por todos los integrantes del equipo, porque la idea es evitar las interpretaciones individuales-disciplinares de los datos-información-conocimiento, con el objetivo de sintetizarlos en una representación única. Para lograrlo deben involucrar las dimensiones y disciplinas que tienen representatividad en el contexto y domino del problema, lo mismo que en la posible solución. Es decir, se debe integrar los datos-información-conocimiento internos y externos para construir un modelo de conocimiento de los requisitos amplio y unificado, involucrando los aportes desde las disciplinas y las dimensiones. De esta manera se adquiere un aprendizaje integrado de los requisitos, que facilita la interpretación que comparte el equipo y las partes interesadas.

En esta integración multidimensional-transdisciplinar del origen de los requisitos participan personas, conocimientos y tecnologías provenientes de diferentes dominios y contextos en los que el sistema existe, o con los que tiene relación. Por lo que se debe: 1) encontrar un lenguaje común necesario para dialogar sobre los requisitos, las opciones, los caminos y el contenido del documento de elicitación. Para esto es importante conformar un equipo de trabajo transdisciplinar, porque se progresa paulatinamente en la comprensión de cada lenguaje hasta el momento de integrar y unificar el conocimiento de los requisitos. 2) Diseñar una metodología para lograr la integración y unificación de los requisitos. Esto no se debe confundir con el intento de establecer el conocimiento de los requisitos como un todo, porque sería como volver a parcelar los datos-información-conocimiento en cada dimensión y disciplina origen.

Para este marco de trabajo un requisito dimensional está definido por un concepto de finalización y un camino de propiedades, es decir, representa una característica compuesta del problema. Debido a que también posee interrelaciones desde o hacia otras dimensiones y disciplinas, hay que considerar los caminos desde un punto de vista multidimensional-transdisciplinar, porque de esta manera se añade la semántica pertinente para su interpretación.

Pero hay que ser cautelosos, porque esas relaciones se pueden dar por medio de *n* diferentes caminos, lo que da lugar a *n* diferentes perspectivas de análisis e interpretación. Además, el equipo debe estar atento, porque todos estos caminos pueden, potencialmente, identificar diferentes conjuntos de instancias en el concepto de finalización. La multidimensionalidad de los requisitos se representa en la Figura 2.

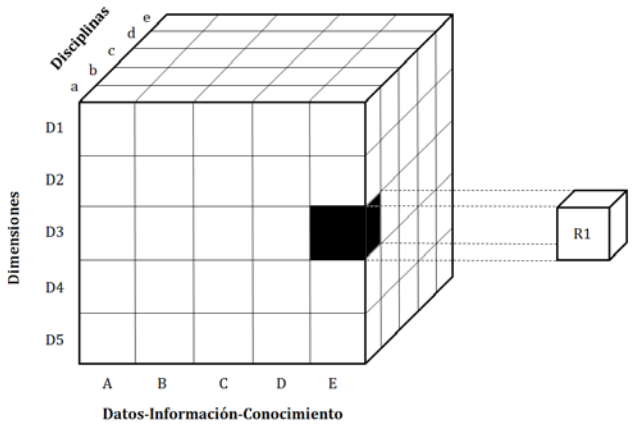

Figura 2. Multidimensionalidad de los requisitos

En esta visión de requisitos multidimensionales no se presume que provienen de una única fuente origen, sino que se caracteriza por representarlos como si se ubicaran en un espacio *n*-dimensional y *n*-disciplinar, en el que es posible comprenderlos y analizarlos fácilmente en términos de los datos-información-conocimiento que se disponen desde diferentes dimensiones y disciplinas, por lo que reflejan diversas fuentes de origen. Por ejemplo, los datos-información-conocimiento necesarios para comprender el requisito 1 (*R1*) provienen de la disciplina a (*a*), que está inmersa en la dimensión 3 (*D3*). En un problema real sería:

Requisito R1: *El sistema debe permitir el acceso de 40 usuarios de forma simultánea*
Datos-Información-Conocimiento: *ancho de banda, tráfico de red, prioridad de los subsistemas, sistema de seguridad*, …
Dimensión D3: *Dimensión Tecnológica* (Interrelación: *Dimensión Administrativa*)
Disciplina a: *Telecomunicaciones* (Interrelación: *Redes de datos*)

Este paradigma ofrece una visualización de requisitos amigable, fácil de entender e intuitiva para el equipo de trabajo. Es importante destacar que en la vida real los problemas presentan este tipo de interrelaciones, por lo que son susceptibles de ser analizados desde un punto de vista multidimensional para comprenderlos y solucionarlos. La Figura 2 se refiere a la colocación de los datos-información-conocimiento en un espacio multidimensional-transdisciplinar, por lo tanto, puede ser interpretado como una *función matemática*. Actualmente, también es común referirse a estos espacios multidimensionales como *cubos de datos*. Sin embargo, hay que tener en cuenta que en la elicitación esta ubicación multidimensional-transdisciplinar, de la cadena datos-información-conocimiento, solamente se utiliza para encontrar el origen multidimensional de los requisitos.

4. CONCLUSIONES

En la elicitación de requisitos los analistas deben valerse de su ingenio para encontrar los requisitos del sistema en las diferentes dimensiones y disciplinas origen. Es un trabajo detectivesco que los conduce a identificar una manera de resolver cada problema específico. En la elicitación tradicional se asume que los requisitos se originan en una única fuente, pero la complejidad y el tamaño de los problemas hacen que las necesidades del cliente, del usuario y del sistema tengan múltiples orígenes, además, que tengan interrelaciones y dependencias que no se pueden rastrear desde una fuente predeterminada.

En este capítulo se propone que la elicitación de requisitos debe ser multidimensional, porque la realidad demuestra que una interrelación o dependencia, que no se tenga en cuenta, puede hacer

que el requisito se pase por alto o que se interprete erróneamente. Como un principio del Pensamiento Complejo, la Multidimensionalidad puede aportarles a los analistas una herramienta que les permita innovar la manera como llevan a cabo la elicitación. También puede ser un aporte que facilite identificar los diferentes caminos relacionales que generan volatilidad y ambigüedad en la interpretación de los requisitos.

Otra cuestión que se concluye de este trabajo es que el proceso de elicitación de requisitos multidimensionales debe iniciarse con la conformación de un equipo transdisciplinar, porque para solucionar los problemas actuales se debe involucrar a personas con visiones y perspectivas diferentes, que pueden confluir en una única interpretación del contexto y el dominio sobre los que se desarrollan. Además, este aspecto es importante porque, como se indicó antes, los requisitos no tienen una única y definitiva fuente de origen y, para estructurar el documento de la elicitación, se necesita un análisis amplio, el cual se establece desde un lenguaje común transdisciplinar.

REFERENCIAS

[1] Terstine M. (2015). The Progress of Requirements Engineering Research. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 5(1), 18-24.

[2] Kitchenham B. (2003). Procedures for undertaking systematic literature reviews. Joint technical report. Keele University.

[3] Pohl K. (1994). The three dimensions of Requirements Engineering. Information Systems 19(3), 243-258.

[4] Durán A. (2000). Un entorno metodológico de Ingeniería de Requisitos para Sistemas de Información. Disertación doctoral. Universidad de Sevilla.

[5] Sawyer P. y Kotonya G. (2001). Software Requirements. En IEEE (Ed.), Guide to the Software Engineering Body of Knowledge. IEEE Computer Society Press.

[6] Silva F. et al. (2002). Towards a Methodology for Requirements Analysis of Data Warehouse Systems. En XVI Simpósio Brasileiro de Engenharia de Software. Rio Grande do Sul, Brasil.

[7] Abelló A. et al. (2000). Benefits of an object oriented multidimensional data model. Lecture Notes in Computer Science 1944, 141-152.

[8] Tuunanen T. (2003). A new perspective on requirements elicitation methods. Journal of Information Technology Theory and Application 5(3), 45-62.

[9] Nurmuliani N. et al. (2004). Using card sorting technique to classify requirements change. En 12th IEEE International Requirements Engineering Conference. Kyoto, Japan.

[10] Coleman D. (2005). Dimensions of interactive software requirements: Synergistic opportunity. En IEEE Southeast Conference. Fort Lauderdale, USA.

[11] Moreira A. et al. (2005). Multi-Dimensional separation of concerns in Requirements Engineering. En 13th IEEE International Conference on Requirements Engineering. La Sorbonne, France.

[12] Annoni E. et al. (2006). Towards multidimensional requirement design. Lecture Notes in Computer Science 4081, 75-84.

[13] Liaskos S. et al. (2007). Exploring the dimensions of variability: A Requirements Engineering perspective. En First International Workshop on Variability Modelling of Software-Intensive Systems. Limerick, Ireland.

[14] Dufresne S. (2008). A Hierarchical Modeling Methodology for the Definition and Selection of Requirements. Doctoral dissertation. Georgia Institute of Technology.

[15] Kano N. et al. (1984). Attractive quality and must-be quality. The Journal of the Japaneses Society for Quality Control 14(2), 39-48.

[16] Pa N. y Zin A. (2011). Requirement Elicitation: Identifying the Communication Challenges between Developer and Customer. International Journal on New Computer Architectures and Their Applications 1(2), 371-383.

[17] Tran H. y Anvari F. (2016). A five-dimensional requirements elicitation framework for e-Learning systems. International Journal of Information and Electronics Engineering 6(3), 185-191.

[18] Serna E. (2012). Analysis and selection to requirements elicitation techniques. En 7th Colombian Computing Congress. Medellin, Colombia.
[19] Gilb T. (2005). Competitive engineering: A handbook for systems engineering, requirements engineering and software engineering management using planguage. Elsevier.
[20] Zave P. (1997). Classification of research efforts in requirements engineering. ACM Computing Surveys 29(4), 315-321.
[21] Viller S. y Sommerville S. (1999). Social analysis in the requirements engineering process: From ethnography to method. En 4th International Symposium on Requirements Engineering. Limerick, Ireland.
[22] González B. et al. (2006). Eliciting non-functional requirements interactions using the personal construct theory. En 14th IEEE International Requirements Engineering Conference. Minneapolis, USA.
[23] Hickey A. et al. (1999). Establishing a foundation for collaborative scenario elicitation. The DATA BASE for Advances in Information Systems 30(3-4), 92-110.
[24] Avison D. y Fitzgerald G. (2002). Information systems development: Methodologies, techniques and tools. McGraw-Hill.
[25] Dafoulas G. y Macaulay L. (2001). Investigating cultural differences in virtual software teams. The Electronic Journal of Information Systems in Developing Countries 7(4), 1-14.
[26] Eveleens J. y Verhoef C. (2010). The rise and fall of the Chaos report figures. IEEE Software 27(1), 30-36.
[27] Lamsweerde A. (2000). Requirements engineering in the year 00: A research perspective. En 22nd International Conference on Software Engineering. Limerick, Ireland.
[28] Boehm B. (2000). The art of expectations management. Computer 33(1), 122-124.
[29] Briggs R. y Gruenbacher P. (2002). EasyWinWin: Managing complexity in requirements negotiation with GSS. En 35th Hawaii International Conference on System Sciences. Hawaii, USA.
[30] Eberlein A. y Leite J. (2002). Agile requirements definition: A view from requirements engineering. En International Workshop on Time-Constrained Requirements Engineering. Essen, Germany.
[31] Moløkken K. y Jørgensen M. (2003). A review of software surveys on software effort estimation. En International Symposium on Empirical Software Engineering. Kaiserslautern, Germany.
[32] Morin E. (2006). Restricted complexity, general complexity. En Intelligence de la complexité: Épistémologie et pragmatique colloquium. Cerisy-La-Salle, France.

CAPÍTULO XX
Estado actual de la investigación en requisitos no-funcionales[1]

Edgar Serna M.
Instituto Antioqueño de Investigación

Generalmente se reconoce que los requisitos no-funcionales son una parte importante y difícil del proceso de la Ingeniería de Requisitos, y que desempeñan un papel fundamental en el desarrollo de productos software. El objetivo de este trabajo es identificar las investigaciones actuales alrededor de la administración de estos requisitos, para lo cual se realizó una revisión sistemática de la literatura para identificar los estudios empíricos disponibles acerca de la temática. Se hizo una búsqueda en bases de datos, y se identificaron 1560 artículos, de los cuales 18 resultaron ser estudios de investigación empírica de alta calidad y relevantes para las preguntas de investigación. También se investigó acerca del concepto que se tiene actualmente de los beneficios y las limitaciones de los métodos para administrar estos requisitos. Se presenta el estado de la investigación en cinco áreas: elicitación, dependencias, métricas, estimación de costos y priorización.

[1] Publicado en la Revista Ingeniería y Universidad 16(1), 225-246. 2012.

INTRODUCCIÓN

La complejidad de los sistemas software está determinada por la funcionalidad y por los aspectos de calidad: rendimiento, fiabilidad, exactitud, seguridad y usabilidad [1]. Estos aspectos se conocen como requisitos no-funcionales del software, y comúnmente se acepta que su manejo y equilibrio es una parte importante y difícil del proceso de la Ingeniería de Requisitos [2], y que juegan un papel fundamental en el desarrollo de software [3].

Una de las características de estos requisitos es la especificación de ciertos niveles de calidad, y, por consiguiente, en muchos casos es posible cuantificarlos [4]. Esto es importante, no solo para su comprensión [2], sino también para su planificación [5]. Tratarlos ineficazmente o no tratarlos puede dar lugar a un producto más costoso, y posiblemente que se demore su salida al mercado [6] o, en el peor de los casos, a errores en el desarrollo del producto [7, 8]. Varios estudios han demostrado que elicitar estos requisitos es costoso y difícil de manejar [6, 9], y de acuerdo con Chung et al. [3], a menudo son mal comprendidos en comparación con aspectos menos críticos del desarrollo de software.

Generalmente se reconoce que las decisiones acerca de cuáles criterios de calidad deben precisarse en un producto tienen grandes efectos en su desarrollo, y en la elección de la arquitectura. Esto significa que el área de los requisitos no-funcionales es importante para comprender con más detalle qué dependencia existe entre su calidad y otros componentes del sistema a desarrollar.

A medida que crece la concientización acerca de los requisitos no-funcionales, también lo hace la literatura acerca de la investigación acerca de los diversos mecanismos, desafíos y estrategias para su administración. Sin embargo, no se encuentra esfuerzos sistemáticos significativos para identificar, sintetizar y reportar la literatura acerca de ellos. Para cubrir esta laguna de investigación, esta revisión sistemática de la literatura tiene por objeto recopilar y comparar la evidencia empírica existente en relación con los requisitos no-funcionales, con la idea de ofrecerles a los investigadores un camino para futuras investigaciones, y a los profesionales una guía para la adopción de tecnologías adecuadas. En esta revisión solamente se investiga los artículos centrados en los métodos y la administración de estos requisitos, es decir, que los trabajos sobre priorización de requisitos en general están por fuera del alcance de esta revisión.

1. MÉTODO

Esta investigación se llevó a cabo como una revisión sistemática de la literatura [10], una forma metódica de identificar, evaluar y analizar la investigación disponible, relevante para una pregunta de investigación particular; es decir, se presenta los elementos de un estudio de mapeo sistemático. Además, incluye actividades como planeación de la revisión, identificación de las preguntas de investigación, estrategia de búsqueda y registro, selección de los estudios, evaluación de la calidad, y extracción y síntesis de los datos.

Se identificaron cinco áreas importantes para la administración de los requisitos no-funcionales: elicitación, métricas, dependencias, estimación y priorización de costos. Debido a que la elicitación está incluida desde el primer paso, identificar los requisitos no-funcionales es importante para cuantificar, es decir, para hacer medibles los criterios de calidad (métricas). Para identificar las interdependencias (dependencias) entre requisitos no-funcionales, es importante estimar antes el costo (estimación de costos). La implementación de un determinado requisito no-funcional desde otro puede tener un efecto negativo o positivo sobre el costo de la implementación de un

segundo requisito no-funcional. Por último, se prioriza los requisitos no-funcionales. En esta investigación se busca responder a las siguientes preguntas:

PI1: ¿Qué se conoce de la investigación empírica sobre los requisitos no-funcionales en relación con la elicitación, la priorización, la estimación de costos, las dependencias y las métricas?

PI2: ¿Qué métodos de investigación empírica se utilizan para evaluar los requisitos no-funcionales?

1.1 Estrategia de búsqueda y registro

La estrategia de búsqueda incluyó la revisión en bases de datos electrónicas y búsquedas manuales en actas de congresos. Se utilizaron las bases de datos: ACM Digital Library, Engineering Village, IEEE Xplore y Wiley Inter Science Journal Finder

Además, la cadena de búsqueda se aplicó a dos bases de datos electrónicas más: ScienceDirect-Elsevier y SpringerLink. Sin embargo, la cadena era demasiado larga y compleja, por lo que se utilizó un sub-conjunto de ella en ambas bases de datos, y se compararon los resultados con las bases de datos iniciales.

Todos los artículos del sub-grupo que se encontraron en ScienceDirect-Elsevier y SpringerLink también se hallaron en las bases de datos primarias. Con base en esta búsqueda de prueba se llegó a la conclusión de que los artículos de ScienceDirect-Elsevier y SpringerLink también serían encontrados en las otras bases de datos.

La International Requirements Engineering Conference, un evento en Ingeniería de Requisitos con reconocimiento en el mundo, está indexada en el Compendex y en la base de datos Inspec, por lo tanto, no se realizaron búsquedas manuales en ella; sin embargo, sí se realizaron en las siguientes:

- International Workshop on Software Product Management IWSPM
- Measuring Requirements for Project and Product Success MeReP
- Requirements Engineering: Foundation for Software Quality REFSQ

Se realizó una búsqueda manual en la conferencia REFSQ debido a que es una de las conferencias más grandes con un único objetivo: la Ingeniería de Requisitos; en IWSPM se buscó manualmente debido a que su enfoque es la gestión de productos software, que incluye la priorización de requisitos; además, se buscó en MeReP debido a que su enfoque son las métricas y mediciones. El proceso de selección de los estudios se realizó en cuatro fases. En la primea se realizó la búsqueda utilizando los términos que se enumeran en la Tabla 1.

Chung et al. [3] definen alrededor de 160 términos diferentes para requisitos no-funcionales, además de varios estándares que también los definen. Para incluirlos todos sería necesario crear una cadena de búsqueda muy larga y complicada, y aun así no se garantizaría que todos fueran cubiertos, por lo que se excluyeron términos específicos acerca de requisitos no-funcionales, como requisitos de desempeño y usabilidad.

Por lo tanto, la categoría C1 quedó compuesta por diferentes términos generales, las categorías C3 y C6 se basaron en revisiones sistemáticas previas [8, 11]. La cadena de búsqueda final, que arrojó una población de 2647 artículos, de los cuales 1560 eran relevantes pata la investigación, se construyó de la siguiente manera: C1 AND C2 AND (C3 OR C4 OR C5 OR C6 OR C7).

Tabla 1. Términos de búsqueda identificados

Categoría	Palabras buscadas	Conector
C1: QR	Non functional requirements Nonfunctional requirements Non-functional requirements Non-functional software requirements Qualities Quality attributes Quality characteristics Quality factors Quality requirements	OR
C2: Software	Software	
C3: elicitación	Elicitation Requirements gathering Requirements acquisition	OR
C4: dependencia	Change impact Conflict Dependenc* Interdependenc* Inter-dependencies Relationships Traceability Trade off Trade offs Tradeoff Trade-off Tradeoffs Trade-offs	OR
C5: métricas	Metrics Measurement	OR
C:5 costos	Software development effort Cost estimation	OR
C7: priorización	Prioritisation Prioritise Prioritising Prioritization Prioritize Prioritizing	OR

1.2 Selección de los estudios

Los 1560 estudios relevantes de la fase 1 se introdujeron en una herramienta de base de datos de referencia. Solo se incluyeron trabajos escritos en inglés. Con el objetivo de determinar la pertinencia para la revisión, en la segunda fase se clasificaron todos los trabajos con base al primer autor. En esta fase fueron excluidos los trabajos con títulos que indicaran claramente que estaban por fuera del alcance de la investigación. Para minimizar la amenaza de excluir algunos relevantes, de los excluidos inicialmente se seleccionaron al azar dos conjuntos muestrales: uno organizado por el primer autor con diferentes aportes, y otro por el segundo y el tercer autor.

Dos documentos fueron objeto de debate, sin embargo, ninguno se añadió a los previamente seleccionados. Después de la fase 2 se identificaron 727 artículos como estudios relevantes. Durante la tercera fase se revisaron los resúmenes de todos los trabajos excluidos por el primer autor, y el 20% de los mismos por el segundo y el tercero. Cuatro trabajos fueron objeto de debate en esta fase, de los cuales se añadió uno a los seleccionados. Al final de la fase quedaron 229 aportes para el proceso de selección de la fase cuatro.

1.3 Evaluación de la calidad

Se utilizaron los criterios de selección propuestos por Dybå y Dingsyr [12], para garantizar la calidad de los trabajos y para excluir aquellos trabajos de investigación no-empírica:

- CS1: El estudio se basa en una investigación empírica
- CS2: Las preguntas, objetivos y propósitos de investigación del estudio están bien definidos
- CS3: El contexto del estudio está bien definido

Cada uno de estos criterios fue calificado en una escala dicotómica: 1: Sí y 0: No. De los 229 trabajos se seleccionaron 18 para llevar a cabo la evaluación de la calidad con base en los tres criterios de selección. Para aceptar un trabajo se definió que debía estar calificado con Sí en el CS1, y con Sí en CS2 o CS3.

La definición de lo que constituye un estudio de caso no siempre es evidente, por ejemplo, el término estudio de caso aparece en los artículos de investigación en Ingeniería del Software, aunque se utiliza ejemplos demasiado pequeños [13]. Por lo tanto, para CS1 se utilizó la siguiente definición para estudio de caso: *Es un método empírico destinado a investigar fenómenos contemporáneos en su contexto* [10]. Cada uno de los 18 estudios fue evaluado de acuerdo con siete criterios de calidad (Tabla 2), que se basan en la lista de verificación para estudios de caso propuesta por Runeson y Höst [13].

Tabla 2. Criterios de evaluación para la calidad

ID	Criterio de calidad
CC1	El diseño de la investigación es adecuado para hacerles frente a las preguntas de investigación
CC2	Los procedimientos de recopilación de datos son suficientes para el propósito
CC3	Los procedimientos de análisis son suficiente para el propósito
CC4	Los resultados están claramente definidos y son creíbles, y las conclusiones están justificadas
CC5	Se adopta diferentes puntos de vista en el caso
CC6	Las amenazas a la validez de los análisis se abordan de manera sistemática
CC7	Las conclusiones son implicaciones para la práctica

Los siete criterios de calidad de la Tabla 2, junto con los tres criterios de selección proporcionan una medida de la calidad de los estudios incluidos, y una medida del grado de confianza en los resultados de cada estudio. En la Tabla 3 se muestra todos los criterios de calidad, clasificados en la misma escala dicotómica de los tres criterios de selección; las áreas se refieren a las cinco áreas identificadas en relación con la pregunta de investigación PI1.

Tabla 3. Valoración de la calidad de los trabajos incluidos

Estudio incluido	Área	CS1	CS2	CS3	CC1	CC2	CC3	CC4	CC5	CC6	CC7	Total
[14]	Priorización	1	1	1	1	1	1	1	1	1	1	10
[2]	Métricas	1	1	1	1	1	1	1	1	1	1	10
[6]	Elicitación	1	1	1	1	1	1	1	1	0	1	9
[15]	Estimación de costos	1	1	1	1	1	1	1	1	0	1	9
[16]	Elicitación	1	1	1	1	1	1	1	1	0	1	9
[17]	Elicitación	1	1	1	1	1	1	1	1	0	1	9
[18]	Elicitación	1	1	1	1	1	1	1	1	0	1	9
[4]	Dependencias	1	1	1	1	1	1	1	1	0	1	9
[5]	Dependencias	1	1	1	1	1	1	1	1	0	1	9
[19]	Dependencias	1	1	1	1	1	1	1	1	0	1	9
[20]	Dependencias	1	1	1	1	0	1	1	1	1	1	9
[21]	Dependencias	1	1	1	1	1	1	1	1	0	1	9

[22]	Métricas	1	1	1	1	1	1	1	0	1	1	9
[23]	Priorización	1	1	1	1	1	1	1	1	0	1	9
[24]	Priorización	1	1	1	1	1	1	1	1	0	1	9
[1]	Elicitación	1	1	1	1	1	0	1	1	0	1	8
[25]	Métricas	1	0	1	1	1	0	1	1	0	1	7
[26]	Métricas	1	1	1	0	0	0	1	1	0	1	6

1.4 Extracción y síntesis de los datos

Se utilizó una forma de extracción de datos predefinida para extraer la información necesaria. De cada estudio se extrajeron los siguientes datos: identificación de la publicación, información bibliográfica (autor(s), año, título e información de la publicación), objetivos y preguntas de investigación, metodología (diseño del estudio, descripción de la muestra, contexto del estudio, recopilación de datos, análisis de datos), herramientas/enfoques/técnicas utilizados, resultados, conclusiones y limitaciones. Se sintetizaron los datos para identificar los temas que se derivan desde los hallazgos reportados en cada uno de los trabajos revisados.

2. RESULTADOS

2.1 PI1: ¿Qué se conoce de la investigación empírica sobre los requisitos no-funcionales en relación con la elicitación, la priorización, la estimación de costos, las dependencias y las métricas?

- *Elicitación*: En la Tabla 4 se muestra las cinco técnicas de elicitación que se han identificado para elicitar requisitos no-funcionales. En los estudios de la muestra se aprecia que no existe una visión clara acerca de cómo elicitar estos requisitos. Todas las técnicas, a excepción de TE3, han sido evaluadas en más de un tipo de sistema.

Tabla 4. Técnicas de elicitación identificadas

Técnica	Origen	Descripción
TE1	[6]	Uso de un léxico
TE2	[1]	Uso de un modelo de calidad, lista de chequeo y un cuestionario de priorización
TE3	[16]	Técnica de la entrevista
TE4	[17, 18]	Estrategia basada en cuestionario soportado en una herramienta
TE5	[17, 18]	Entrevistas estructuradas

En los estudios de la muestra se encontraron algunas sugerencias acerca de cómo elicitar requisitos no-funcionales. Por ejemplo, Cysneiros y Leite [6] sostienen que estos requisitos no deberían tratarse con el mismo alcance de los requisitos funcionales, debido a que requieren un razonamiento más detallado; Doerr et al. [1] argumentan que la elicitación de los requisitos funcionales, los no-funcionales y la arquitectura debe estar ligada, debido a que el refinamiento de los no-funcionales no es posible sin el detalle de los funcionales y la arquitectura; además, Hassenzhal et al. [16] afirman que es importante reunir diferentes aspectos, tales como requisitos no-funcionales, enfoque de diseño y las relaciones entre ellos, para asegurar una comprensión básica del problema a diseñar. El punto de vista de Hassenzhal et al. [16] acerca de la relación entre requisitos no-funcionales, el diseño y sus relaciones, es soportado por Cysneiros y Leite [6], que establecen que los no-funcionales tienen muchas interdependencias entre sí que pueden requerir *negociaciones* entre las diferentes decisiones de diseño.

En términos del logro de objetivos, las diferentes técnicas de elicitación resultaron ser prometedoras, y elicitaron más requisitos no-funcionales que otras técnicas utilizadas

anteriormente en los estudios de caso. TE1 genera entre 20% y 25% de nuevas clases, y 46% de las clases existentes se cambiaron para satisfacer los requisitos no-funcionales elicitados. Doerr et al. [1] sostienen que con TE2 se descubrieron nuevos e importantes requisitos no-funcionales, y que solo cinco de los 54 elicitados no eran medibles. Con TE3 se encontraron 172 cualidades concretas, sin embargo, no se encontró información acerca del porcentaje de nuevos requisitos no-funcionales elicitados. Para TE5 y TE6 la elicitación dio lugar a una serie de cambios en las actividades de desarrollo previstas.

- *Dependencias*: La Tabla 5 muestra los tres métodos de dependencia MD que se ha estudiado empíricamente en relación con los requisitos no-funcionales. Los tres métodos cuentan con el soporte de herramientas que se utilizan en su evaluación empírica. Además de los métodos identificados, en [22] también se investiga acerca de la existencia de interdependencias entre los requisitos no-funcionales.

Tabla 5. Métodos de dependencia identificados

MD	Origen	Descripción
MD1	[4]	QARCC
MD2	[19, 20]	QARCC y S-COST
MD3	[5]	Trazabilidad centrada en objetivos

Zulzalil et al. [21] analizan las interdependencias entre los requisitos no-funcionales para tres aplicaciones web diferentes. En dos de ellas encontraron una fuerte dependencia entre la usabilidad y la eficiencia, la funcionalidad y eficiencia, y usabilidad y fiabilidad. Por otra parte, en una de las aplicaciones encontraron una fuerte dependencia entre la usabilidad y la funcionalidad, y entre la funcionalidad y la fiabilidad.

Al igual que en [21], Boehm e In [4], In et al. [19] e In y Boehm [20] analizan los conflictos entre los requisitos no-funcionales. Sin embargo, en estos estudios se evaluaron los métodos MD1 y MD2 para identificarlos. Cleland et al. [5] también evaluaron el método MD3, sin embargo, su enfoque es el impacto del cambio funcional en los requisitos no-funcionales. Los tres métodos fueron evaluados y comparados con los enfoques manuales: MD1 y MD2 con el uso manual del modelo WinWin [20], mientras que MD3 con el número de enlaces que se requiere en un análisis manual. Todos los métodos evaluados mostraron resultados prometedores en comparación con los manuales, como se ilustra en la Tabla 6.

Tabla 6. Eficiencia de las herramientas de dependencias

MD	Enfoque manual	Herramienta
MD1	2 conflictos encontrados	10 conflictos encontrados
MD2	7 conflictos encontrados	76 conflictos encontrados
MD3	360 posibles comparaciones	22 enlaces para analizar

Con MD1 y MD2 se identificaron más conflictos de calidad en comparación con el uso manual de WinWin. En el estudio de In et al. [19] se identificaron manualmente siete conflictos de calidad, y 76 con MD2 –24 insignificantes. Además, con MD2 se identificaron cinco de los siete conflictos encontrados con el enfoque manual; con MD1 y MD2 se identificaron los mismos conflictos hallados con el uso manual de WinWin. Además, con la herramienta identificaron ocho conflictos más, de los cuales cinco fueron significativas. Con MD3 evaluaron la capacidad de los métodos, para reducir el número de enlaces para gestionar el impacto del cambio funcional sobre los requisitos no-funcionales. Mientras que un análisis manual requiere 360 enlaces para ser analizados [5], MD3 reduce ese número a 22.

Con base en los resultados empíricos en esta revisión sistemática, se encontró que las herramientas les ayudan a desarrolladores y administradores a tratar las dependencias de forma más eficiente mediante la identificación de más conflictos, o reduciendo el número de enlaces necesarios para examinar los cambios en el sistema.

- *Métricas:* De acuerdo con Jacobs [26], es necesario establecer requisitos medibles. Uno de los objetivos de la técnica de elicitación TE2 es lograr un conjunto de requisitos no-funcionales medibles, y casi el 90% de los elicitados con la técnica fueron cuantificados. Por otro lado, cuando Olsson et al. [22] investigaron una especificación de requisitos de un estudio de caso, donde el 40% de los 2113 requisitos eran no-funcionales, fue posible cuantificar cerca de la mitad de ellos. También analizaron la forma en se cuantifica los requisitos no-funcionales, y encontraron que se utiliza una mezcla de diferentes escalas:

  - Absoluta: Sin intervalo dado (58%)
  - Min-Max: Se especifica un límite inferior y superior, creando un intervalo min-max (7%)
  - Límite superior: Intervalo unilateral (24%)
  - Límite inferior: Intervalo unilateral (12%)

  Por otra parte, manifiestan que la naturaleza de los intervalos y escalas especificados para las diferentes áreas, incluso dentro del mismo producto software, son importantes para la negociación y la priorización. Es decir, intervalos y escalas deben estar alineados con el mercado y el valor del costo.

  En esta investigación se identificaron dos enfoques para cuantificar requisitos no-funcionales: QUPER [2, 25] y el estilo Gilb [26]. La importancia de los enfoques específicos para cuantificar estos requisitos es apoyada por Jacobs [26], quien afirma que es más difícil y esencial cuantificar requisitos no-funcionales que funcionales. El modelo QUPER considera la alineación entre los intervalos y los valores de mercado sugerido por Olsson et al. [22].

  Los dos enfoques tienen características similares, ambos tienen conceptos para identificar mediciones para los productos de los proveedores y lo que el mercado espera. Además, ambos cuantifican requisitos no-funcionales para un intervalo, similar al Min-Max en Olsson et al. [22]. Sin embargo, el intervalo Min-Max es la escala menos utilizada, pero lo interesante es que tanto QUPER como el estilo Gilb sugieren el uso de intervalos.

  Ambos enfoques fueron evaluados suficientemente. En QUPER la relación entre los intervalos y el valor de mercado es visto como una característica importante, y ajustando el intervalo de requisitos no-funcionales, con base al segmento de mercado y la calidad de los competidores, se proporciona una mejor base para las métricas reales. Al introducir el estilo Gilb se logra una comprensión común de los requisitos no-funcionales, lo que, de acuerdo con Jacobs [26], es crucial. Por otra parte, mediante la especificación de los conceptos del estilo Gilb para cada requisito no-funcional, Jacobs encontró que los casos de prueba se pueden definir desde la fase de la Ingeniería de Requisitos.

- *Estimación de costos:* Solo Regnell et al. [15] relaciona la estimación del costo de los requisitos no-funcionales. Sin embargo, no se refieren a cómo se lleva a cabo en la industria esta estimación; en su lugar identifican cómo prever las barreras de los costos. Evalúan si es posible prever cuándo es necesaria una inversión importante (en términos de costos) para mejorar el nivel de calidad; encontraron que el costo de alcanzar un determinado nivel de calidad está relacionado con la optimización del software, la inversión en hardware, el esfuerzo de

desarrollo, las inversiones en nueva arquitectura, y los derechos de las licencias. Además, que la estimación de costos es más incierta en las primeras etapas de una nueva tecnología, en comparación con una que ya está disponible y que alcanzó la madurez en un mercado determinado.

- *Establecimiento de prioridades:* Tres estudios relacionan la priorización de los requisitos no-funcionales más importantes [14, 23, 24]. El análisis hecho a los datos extraídos puso de manifiesto que ninguna técnica de priorización, que se centre únicamente en los requisitos no-funcionales, se ha evaluado empíricamente. Sin embargo, en general las técnicas de priorización de requisitos se pueden utilizar a la hora de priorizarlos, por lo tanto, están fuera del alcance de esta revisión. [23] y [24] abordan cómo las diferentes partes interesadas priorizan la importancia de los requisitos no-funcionales, mientras que [14] desarrolla y evalúa un marco de proceso para priorizar esa importancia.

En esta revisión se encontró que las diferentes partes interesadas tuvieran diferentes puntos de vista con respecto a la importancia de los requisitos no-funcionales, tanto entre las de dentro de la misma empresa, donde los objetivos son los mismos [23], como entre los diferentes dominios [14, 23]. Johansson et al. [23] encontraron que la fiabilidad era identificada por una multitud de interesados, para enfocarla en los más importantes requisitos no-funcionales; [24]) afirma que era identificada como el requisito funcional más importante para las aplicaciones de intranet; mientras que [14] informan que la funcionalidad, para sus dos proyectos, es uno de los requisitos no-funcionales más importante.

Los resultados revelan que, dentro de la misma organización, las partes interesadas, tales como mercadeo, diseñadores de sistemas y arquitecturas, pueden tener diferentes puntos de vista acerca de a cuáles requisitos no-funcionales se deba considerar importantes [23]. Una explicación puede ser, por ejemplo, que los arquitectos y los diseñadores de sistemas trabajan en diferentes niveles de la arquitectura [23].

Para ayudarles a las organizaciones a personalizar un modelo de calidad del software para sus necesidades específicas, Sibisi y van Waveren [14] desarrollaron un marco de procesos para vincular, desde el estándar ISO/IEC 9126-1, a las necesidades del usuario, las características de calidad y sus sub-características. El resultado de validación revela que el marco es válido a nivel de las características, sin embargo, no se encontró un estudio o evaluación acerca del tiempo necesario para aprender y calibrar el marco para necesidades específicas de cada organización.

2.2 PI2: ¿Qué métodos de investigación empírica se utilizan para evaluar los requisitos no-funcionales?

En la Tabla 7 se observa que las dos terceras partes de la muestra (12 trabajos) incluyen el uso de estudios de casos como método de investigación para evaluar empíricamente los requisitos no-funcionales, lo que lo convierte en el método de investigación más común. Siete de ellos utiliza un solo caso, mientras que los otros cinco utilizan múltiples casos.

Tabla 7. Distribución de los métodos de investigación

Método	Publicaciones
Estudio de caso	12 (7 uno solo, 5 múltiples)
Experimento	3
Encuesta	2
Mixto (encuesta y experimento)	1

Se podría argumentar que esto indica que existe una necesidad de evaluar los requisitos no-funcionales y sus técnicas, utilizando otros métodos de investigación diferentes al estudio de caso, pero solo Seaman [27] apoya la idea. Este autor argumenta que las cuestiones de la Ingeniería del Software son las más investigadas, pero que para ello se utiliza una combinación de métodos cualitativos y cuantitativos. Sin embargo, mirando qué métodos de investigación se ha utilizado para cada una de las categorías identificadas en esta investigación en relación con los requisitos no-funcionales, los resultados muestran que todos los estudios, en relación con la elicitación y las métricas, han utilizado estudios de caso. Por otra parte, los tres experimentos se llevaron a cabo en relación con las dependencias de los requisitos no-funcionales, mientras que todas las encuestas investigan la priorización de los mismos.

3. ANÁLISIS DE RESULTADOS

El objetivo general de esta investigación es recopilar y comparar la evidencia empírica existente acerca de los requisitos no-funcionales, además de proporcionar el estado del arte que sirva de base para futuras investigaciones empíricas acerca de estos requisitos. A continuación, se analiza los resultados obtenidos.

3.1 Beneficios y limitaciones

Los 18 estudios de la muestra se clasificaron en cinco categorías relacionadas con las actividades de la Ingeniería de Requisitos. La revisión mostró que se lleva a cabo más estudios empíricos acerca de los requisitos no-funcionales en relación con la elicitación y las categorías de dependencias. Los estudios que se centran en la elicitación de estos requisitos no proporcionan un punto de vista unificado de la práctica, en cambio, ofrecen un amplio panorama de las experiencias y las técnicas probadas. En esta revisión sistemática se identificaron cinco diferentes técnicas de elicitación de estos requisitos, cada una se evaluó en uno, dos o tres sistemas diferentes, y y todas resultaron ser prometedoras para elicitar requisitos no-funcionales.

Davis et al. [28] encontraron que la entrevista es la técnica de elicitación más efectiva para recopilar estos requisitos. En esta revisión sistemática, tres de las cinco técnicas utilizan entrevistas para elicitar los requisitos no-funcionales. Por otra parte, los hallazgos de [28] solo se confirman parcialmente en este estudio. Con respecto a las limitaciones, Doerr et al. [1] sostienen que la elicitación de requisitos no-funcionales y funcionales debe estar entrelazada con la arquitectura. Por otra parte, Cysneiros y Leite [29] sostienen que ambos requisitos no deben tratarse dentro del mismo ámbito de aplicación. La razón es que los no-funcionales requieren un razonamiento detallado. Sin embargo, la idea de Doerr et al. [1] apoya el estudio de van Lamsweerde [30], quien identificó la necesidad de cerrar la brecha entre requisitos y arquitectura.

Con respecto a la manipulación de las dependencias de los requisitos no-funcionales se identificaron tres métodos diferentes, aunque en la práctica no se proporciona una visión unificada al aspecto. En los estudios de Boehm e In [4] e In y Boehm [20] se hace una comparación entre el enfoque manual del modelo WinWin con los métodos QARCC, y QARCC en combinación con el método S-COST. Ambos estudios mostraron que el modelo WinWin fue menos eficaz en la identificación de los conflictos entre los requisitos no-funcionales. Sin embargo, no se presenta una comparación entre el método QARCC y la combinación de los métodos QARCC y S-COST. van Lamsweerde [30] sugiere la necesidad de una herramienta de soporte en la Ingeniería de Requisitos. Con respecto a las limitaciones, el modelo WinWin es el único método que ha sido evaluado en más de un estudio, por lo tanto, no fue posible identificar cuál es el método más adecuado para para identificar las interdependencias entre los requisitos no-funcionales.

Con respecto a la priorización de los requisitos no-funcionales se identificaron tres estudios, sin embargo, cabe señalar que ninguno de los de la muestra describe qué técnica o método puede ser más eficiente para priorizarlos, lo que indica la importancia de comprender cómo se priorizan estos requisitos, además, que puede ser un vacío en la literatura.

En esta revisión se identificó un estudio relacionado con los requisitos no-funcionales y la estimación de costos. Una posible explicación es que la cadena de búsqueda no identificó los estudios existentes, y según [11] esto puede explicarse porque en la literatura se utiliza una serie de sinónimos para los términos buscados. Estos investigadores llegan a la conclusión de que para que una búsqueda más amplia tenga sentido se deben incluir más sinónimos, lo que se traduciría en un conjunto más grande de estudios encontrados. Otra explicación puede ser que la estimación de costos de los requisitos funcionales y no-funcionales se realiza de manera similar.

En esta revisión no se encontró ningún estudio en el que se trate la forma de estimar el costo de los requisitos no-funcionales, por lo que, de acuerdo con el conocimiento adquirido en esta revisión y en los resultados de Kitchenham et al. [31], es posible concluir que falta evidencias acerca de cómo se lleva a cabo la estimación del costo de los requisitos no-funcionales. Se podría argumentar que para la estimación de costos, sin importar si es para requisitos funcionales o no-funcionales, se puede utilizar el mismo proceso. Sin embargo, los estudios de Brooks [9] y Cysneiros y Leite [6] han demostrado que los no-funcionales son caros y difíciles de manejar, y, de acuerdo con Chung et al. [3], los no-funcionales son a menudo poco conocidos. Esto indica la importancia de comprender cómo se lleva a cabo el procedimiento de estimación de costos de estos requisitos, lo que también puede ser un vacío en la literatura.

3.2 Fuerza de la evidencia

La discusión de la solidez de las evidencias se basa de en dos categorías: el diseño y la calidad del estudio (Tabla 3). El diseño del estudio de Dybå y Dingsyr [12] se refiere a los métodos de evaluación utilizados, donde los experimentos se consideran la evidencia más fuerte, mientras que los estudios observacionales son evidencias débiles. Además, Seaman [27] argumenta que, para investigar los problemas de Ingeniería del Software, es preferible una combinación de métodos cualitativos y cuantitativos.

12 de los 18 estudios incluidos en la muestra de la revisión son estudios de caso, lo que puede implicar que la evidencia total de los estudios combinados es baja en relación con el diseño del estudio. Con respecto a la calidad de los métodos de los estudios incluidos, a veces no están bien descritos, cuestiones como parcialidad, validez y procedimientos de análisis de datos adecuados no siempre se tratan. Solo en cuatro de los 18 estudios se abordan cuestiones de validez de forma sistemática. Con base en estos resultados, se concluye que existen limitaciones para la calidad de los estudios.

4. CONCLUSIONES

En esta revisión sistemática de la literatura se clasificaron los estudios en cinco áreas en relación con los requisitos no-funcionales: elicitación, dependencias, métricas, estimación de costos, y priorización.

En relación con la pregunta de investigación PI1: ¿Qué se conoce de la investigación empírica sobre los requisitos no-funcionales en relación con la elicitación, la priorización, la estimación de costos, las dependencias y las métricas? los estudios se clasifican en cinco categorías.

Los estudios que tratan la elicitación de los requisitos no-funcionales no proporcionan un punto de vista unificado. Por ejemplo, Cysneiros y Leite [29] argumentan que estos requisitos no deben tratarse en el mismo ámbito de los funcionales, mientras que Doerr et al. [1] y Hassenzahl et al. [16] sostienen que los requisitos no-funcionales y otros aspectos, como el diseño y la arquitectura, deben tratarse de forma conjunta. Se evaluaron cinco técnicas de elicitación diferentes, de las cuales tres se basan en entrevistas. Este resultado está en línea con el trabajo de Davis et al. [28], quienes encontraron que la entrevista es la técnica de elicitación más comúnmente utilizada.

Con respecto al manejo de las dependencias de los requisitos no-funcionales se identificaron tres métodos con el soporte de una herramienta. Estos métodos fueron más eficientes que los enfoques manuales utilizados en la evaluación. Los resultados empíricos de esta revisión indican que las herramientas les ayudan a los profesionales a hacerles frente a las dependencias entre los requisitos no-funcionales, de forma más eficiente que con los métodos manuales. Este resultado está en línea con van Lamsweerde [30], quien apoya la necesidad de una herramienta de soporte en los procesos de la Ingeniería de Requisitos. Sin embargo, no se encontraron comparaciones entre los tres métodos, por lo tanto, no es posible identificar cuál es el más adecuado para para identificar las dependencias entre los requisitos no-funcionales.

Con respecto a las métricas para los requisitos no-funcionales se encontraron dos enfoques con características similares para cuantificarlos. Ambos enfoques sugieren la cuantificación de los requisitos no-funcionales mediante intervalos, sin embargo, Olsson et al. [22] encontraron que un intervalo Min-Max es la escala menos utilizada para cuantificar estos requisitos en la industria, mientras que un valor absoluto es la cuantificación más común.

Se identificó un estudio relacionado con la estimación de costos de los requisitos no-funcionales, sin embargo, no presenta un método o técnica para esa estimación. Una razón puede ser que los métodos de estimación de costos son empíricamente evaluados para requisitos en general y, sin importar si se trata de funcionales o no-funcionales, se puede utilizar el mismo método. Sin embargo, los estudios de Brooks [9] y Cysneiros y Leite [6] muestran que los no-funcionales son más difíciles de manejar.

Ninguno de los estudios relacionados con la priorización tuvo en cuenta técnicas o métodos para priorizar los requisitos no-funcionales. En cambio, la evidencia empírica muestra que las diferentes partes interesadas tienen diferentes puntos de vista con respecto a la importancia de estos requisitos, incluso dentro de la misma empresa, donde los objetivos son los mismos. Además, en dos estudios se identifica a la fiabilidad como el requisito no-funcional más importante, mientras que la calidad, como aspecto de la funcionalidad, es vista como el más importante en un estudio.

Los resultados para la pregunta de investigación PI2: ¿Qué métodos de investigación empírica se utilizan para evaluar los requisitos no-funcionales? muestran que el estudio de caso es el método de investigación más utilizado. Sin embargo, al analizar qué método de investigación se ha utilizado en cada una de las cinco áreas, los resultados son diferentes. Solo los estudios de caso se utilizan en las áreas de elicitación y las métricas, mientras que las encuestas se utilizan con respecto a la priorización de los requisitos no-funcionales.

Con respecto a las limitaciones no se percibe un punto de vista unificado de las prácticas. Pocos estudios son replicados, por lo que la posibilidad de sacar conclusiones con base en variaciones es limitada. Para que los profesionales puedan utilizar estos resultados para seleccionar qué

método o técnica aplicar, deben considerarlos y compararlos con el contexto real en el que el método o la técnica se va a aplicar.

Una conclusión clara de esta revisión es la necesidad de aumentar el número y la calidad de los estudios acerca de los requisitos no-funcionales, por lo tanto, se propone:

- Evaluar empíricamente los requisitos no-funcionales en diversas actividades de Ingeniería de Requisitos para llenar las lagunas identificadas en la literatura.
- La priorización de los requisitos no-funcionales requiere más atención, ya que ni un solo estudio de la muestra examina las técnicas de priorización de estos requisitos.
- Es necesario realizar replicaciones de los estudios en contextos diferentes y ampliarlas a entornos más complejos.

REFERENCIAS

[1] Doerr J. et al. (2005). Nonfunctional requirements in industry - Three case studies adopting an experience-based NFR method. En 13th IEEE International Conference on Requirements Engineering. Paris, France.

[2] Berntsson S. et al. (2008). Introducing support for release planning of quality requirements: An industrial evaluation of the quper model. En Second International Workshop on Software Product Management. Barcelona, Spain.

[3] Chung L. et al. (2000). Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers.

[4] Boehm B. e In H. (1996). Identifying quality-requirements conflict. IEEE Software 12(6), 25-35.

[5] Cleland J. et al. (2005). Goal-centric traceability for managing non-functional requirements. En 27th International Conference on Software Engineering. St. Louis, USA.

[6] Cysneiros L. y Leite J. (1999). Integrating Non-Functional Requirements into Data Model. En Fourth IEEE International Symposium on Requirements Engineering. Limerick, Ireland.

[7] Breitman K. et al. (1999). The World's Stage: A Survey on Requirements Engineering Using a Real-Life Case Study. Journal of the Brazilian Computer Society 6(1), 13-38.

[8] Finkelstein A. y Dowell J. (1996). A Comedy of Errors: The London Ambulance Service Case Study. En Eight International Workshop on Software Specification and Design. Schloss Velen, Germany.

[9] Brooks F. (1987). No Silver Bullet: Essences and Accidents of Software Engineering. IEEE Computer 4, 10-19.

[10] Kitchenham B. (2010). Guidelines for performing Systematic Literature Reviews in Software Engineering. Tech. Rep., EBSE-2007-001. Durham University.

[11] Jørgensen M. y Shepperd M. (2007). A Systematic Review of Software Development Cost Estimation Studies. IEEE Transactions on Software Engineering 31(1), 33-53.

[12] Dybå T. y Dingsyr T. (2008). Empirical studies of agile software development: A systematic review. Information and Software Technology 50(9-10), 833-859.

[13] Runeson P. y Höst M. (2009). Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2), 131-164.

[14] Sibisi M. y Van Waveren C. (2007). A process framework for customising software quality models. En IEEE AFRICON Conference. Windhoek, Namibia.

[15] Regnell B. et al (2007). A Quality Performance Model for Cost-Benefit Analysis of Nonfunctional Requirements Applied to the Mobile Handset Domain. En 13th International Working Conference on Requirement Engineering: Foundation for Software Quality. Trondheim, Norway.

[16] Hassenzahl M. et al. (2001). Exploring and understanding product qualities that users desire. En 5th Annual Conf. of the Human-Computer Interaction Group of the British Computer Society. Lille, France.

[17] Kusters R. et al. (1999). Identifying embedded software quality: two approaches. Quality and Reliability Engineering International 15(6), 485-492.

[18] Kusters R. et al. (1999). Strategies for the identification and specification of embedded software quality. En Ninth Intern. Workshop Software Technology and Engineering Practice. Pittsburgh, USA.

[19] In H. et al. (2001). Applying WinWin to quality requirements: a case study. En 23rd International Conference on Software Engineering. Miami, USA.
[20] In H. y Boehm B. (2005). Using WinWin quality requirements management tools: a case study. Annals of Software Engineering 11(1), 141-174.
[21] Zulzalil H. et al. (2008). Relationships analysis between quality factors for Web applications. En International Symposium on Information Technology. Kuala Lumpur, Malaysia.
[22] Olsson T. et al. (2007). Non-functional requirements metrics in practice: An empirical document analysis. En Workshop on Measuring Requirements for Project and Product Success. Palma de Mallorca, Spain.
[23] Johansson E. et al. (2001). The importance of quality requirements in software platform development-a survey. En 34th Annual Hawaii International Conference on System Sciences. Hawaii, USA.
[24] Leung H. (2001). Quality metrics for intranet applications. Information and Management 38(3), 137-152.
[25] Regnell B. et al. (2008). Supporting roadmapping of quality requirements. IEEE Software 25(2), 42-47.
[26] Jacobs S. (1999). Introducing Measurable Quality Requirements: A Case Study. En Fourth IEEE International Symposium on Requirements Engineering. Limerick, Ireland.
[27] Seaman C. (1999). Qualitative methods in empirical studies of software engineering. IEEE Transactions on Software Engineering 25(4), 557-572.
[28] Davis A. et al. (2006). Effectiveness of Requirements Elicitation Techniques: Empirical Results Derived from a Systematic Review. En 14th IEEE International Requirements Engineering Conference. Minneapolis, USA.
[29] Cysneiros L. y Leite J. (2004). Nonfunctional Requirements: From Elicitation to Conceptual Models. IEEE Transactions on Software Engineering 30(5), 328-349.
[30] van Lamsweerde A. (2000). Requirements Engineering in the Year 00: A Research Perspective. En International Conference on Software Engineering. Limerick, Ireland.
[31] Kitchenham B. et al. (2009). Systematic literature reviews in software engineering - A systematic literature review. Information and Software Technology 51(1), 7-15.

TERCERA PARTE

PRUEBAS DEL SOFTWARE

CAPÍTULO XXI
Acercamiento ontológico a la Gestión del Conocimiento en el mantenimiento del software[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

El concepto de ontologías ha sido ampliamente reportado en la literatura y se ha descrito varios diseños ontológicos, pero pocos trabajos intentan explicar en términos prácticos cómo diseñar una ontología. Por otro lado, aunque se acepta ampliamente la importancia del mantenimiento de los productos software, pocos diseños ontológicos se centran en aplicar técnicas de Gestión de Conocimiento para conseguirlo. En este trabajo se analiza las ontologías propuestas para atender esta necesidad, con el fin de presentar información que pueda ayudar a las organizaciones de desarrollo de software en trabajos similares. Además, se describe un proceso metodológico para estructurar una ontología que se puede aplicar para gestionar el conocimiento en el mantenimiento de software.

[1] Publicado en inglés en International Journal of Information Management 34, 704–710. 2014.

INTRODUCCIÓN

Para realizar trabajos eficientes en los modelos de Ingeniería del Software y aplicarlos en cada una de sus fases, el conocimiento requerido es variado, de grandes proporciones y en incremento constante. La investigación en Ingeniería del Software se orienta hacia la Gestión del Conocimiento, procurando tomar las mejores decisiones y proporcionándoles a las organizaciones la información que requieren para el desarrollo de las fases [1].

La fase de mantenimiento del software es una actividad en la que el conocimiento juega un importante rol; el nivel de conocimiento de los encargados de realizarla es voluminoso, complejo e intensivo en áreas como: el dominio del programa, de la organización que lo utiliza, el pasado y presente de las prácticas de Ingeniería del Software, los diversos lenguajes de programación, la metodología de programación utilizada, las relaciones entre los diferentes módulos y las herramientas necesarias, entre otras [2]. Con frecuencia, la información necesaria para desarrollar este rol no se encuentra o es muy difícil de localizar y reconstruir; por ello, para realizar su labor, los encargados deben realizar consultas en la escasa documentación disponible o a través de compañeros de trabajo, lo que origina que parte del conocimiento existente en los grupos de mantenimiento se pierda o que no se utilice [3].

La Gestión del Conocimiento provee técnicas y métodos que ayudan a reducir la pérdida o el desaprovechamiento de conocimiento, y permite que los encargados del mantenimiento de software puedan compartirlo [4]; y a las organizaciones de desarrollo y mantenimiento de software les asegura beneficios como mejoras de la calidad de sus productos y procesos, y la reducción de costos y errores [5]. Sin embargo, antes de iniciar procesos de desarrollo de sistemas de Gestión del Conocimiento, es importante identificar el conocimiento que se va gestionar, el lugar en el que se almacena y en que se requiere. Además, como generalmente la organización no sabe cómo localizar o cuál es la persona poseedora del conocimiento necesario para resolver el problema en cuestión, también se convierte en tarea por realizar [6].

Para ayudar en la gestión de ese conocimiento los investigadores trabajan desde hace algún tiempo en la conceptualización de ontologías como modelos del dominio, que actualmente emergen como uno de los instrumentos de Gestión del Conocimiento más apropiados para soportar su representación, tratamiento, almacenamiento y recuperación, y que se comienza a extender a todas las fases de la Ingeniería del software. En relación con el mantenimiento, y considerando los tipos de conocimiento involucrados en las ontologías que lo apoyan, los desarrollos ontológicos se nutren de experiencias y aportes diversos, pero pocos llegan a modelar e implementar una ontología representativa de esta área.

1. MARCO REFERENCIAL

1.1 Mantenimiento del software

El mantenimiento de software se define como cualquier modificación de un producto software, después de su entrega, para corregir errores, mejorar el rendimiento u otros atributos, o a la acción de adaptar el producto a un entorno que cambia [7]; cambios en la gestión de productos software para mantenerlos actualizados y en pleno funcionamiento [8]. A este respecto, en el último decenio las técnicas de desarrollo de software avanzaron notoriamente y se propusieron y probaron nuevos procesos, nuevos lenguajes y nuevas herramientas; por el contrario, el mantenimiento de software, al parecer, se quedó a la zaga. Este tema, de mucha pertinencia e importancia en la Ingeniería del Software, recibe relativamente poca atención en la literatura

técnica [9]. Sistematizar el mantenimiento es difícil, fundamentalmente porque es una actividad reactiva y, por tanto, caótica para desarrollar.

Mientras que los proyectos de desarrollo de software pueden durar meses, el mantenimiento, por lo general, dura varios años. La realización de estimaciones de recursos es un elemento clave de la planificación del mantenimiento, y estos recursos se deben incluir en el presupuesto de planificación. La planificación de mantenimiento de software debe comenzar con la decisión de desarrollar un nuevo sistema, y considerar los objetivos de calidad que lEEE recomienda [10]. Esta fase del ciclo de vida del software es el resultado de la necesidad de adaptar los sistemas a un entorno siempre cambiante en el que, en la mayoría de casos, no se puede evitar retrasos.

Las organizaciones deben mantenerse al ritmo de estos cambios, lo que a menudo significa que deben modificar el software sobre el que apoyan sus actividades. Una solución factible para romper este círculo vicioso es el desarrollo de sistemas de Gestión de Conocimiento para el mantenimiento del software, y evitar una práctica común a la mayoría de las organizaciones: intentar volver a documentar sus sistemas software, ya que es una operación costosa en la que se alejan del beneficio de los programas instalados para centrarse en la documentación.

1.2 Ontologías

En el ámbito de las TI el término *ontología* suscita gran interés, especialmente luego de que el World Wide Web Consortium W3C la considerara como la tecnología llamada a facilitar la infraestructura de conocimiento a la Web Semántica o Web 3.0 [11, 12]. Pero, para ser una tecnología cuyo objetivo es clarificar, explicitar y consensuar el conocimiento relacionado con un dominio determinado, es paradójico que no alcance un consenso que ofrezca claridad acerca de lo que es o debería ser [13].

Lacy [14] manifiesta que el término *ontología* se sobrecarga, en tanto que su significado se refiere a diferentes cosas según quién lo defina. Axiomas más tradicionales permiten establecer una diferencia con el concepto filosófico, al determinar que *Ontología* (con mayúscula) es un campo que en filosofía estudia la naturaleza y organización de la realidad o la existencia, en relación directa con la epistemología. Mientras que una de las definiciones más citada en el ámbito de la ingeniería del conocimiento la define como una especificación explícita de una conceptualización [15]. Una ontología es la descripción conceptual y terminológica de un conocimiento compartido acerca de un dominio específico. Dejando de lado la formalización e interoperabilidad de aplicaciones, esto no es más que la principal competencia del término: hacer mejoras en la comunicación utilizando un mismo sistema en lo terminológico y conceptual [16]. De estas concepciones es importante tener en cuenta que, contrario a la filosófica, la ontología debe considerarse no como una entidad natural que se descubre, sino como un recurso artificial que se crea con un objetivo determinado, y para una aplicación concreta [17]. Aceptar esto es aceptar que la aplicación determina, en muchos casos, la categorización que se hace.

Las taxonomías se diseñan desde un punto de vista: el criterio explícito de una jerarquía resultante, que se manifiesta en el resto de relaciones, las cuales definen y dotan de expresividad al conjunto. Debido a esto, diseñar una ontología implica hacer elecciones y seleccionar criterios; y ya que su objetivo es servir como referencia a personas, aplicaciones y organizaciones, esas elecciones y categorizaciones se deben consensuar y reconocer [18]. En el caso de los proyectos de mantenimiento de software es muy útil tener ontologías definidas acerca de la temática de gestión de los mismos, ya que se resuelve estos equívocos y se evita las discusiones originadas en la comprensión del concepto mismo de *petición de mantenimiento* [19].

2. ONTOLOGÍAS ALREDEDOR DEL MANTENIMIENTO DEL SOFTWARE

Antes de estructurar un sistema de gestión del conocimiento para el mantenimiento del software, es importante que se piense en modelar, estructurar y generalizar la información que se genera y se consulta en dicho proceso. Para lograr eficientemente esta actividad se utiliza las ontologías con las que, según Gruber [20], se hace la especificación explícita de una conceptualización. Esta tecnología se puede utilizar para que el conocimiento de la organización pueda compartirse, y para promover la característica de interoperabilidad entre sistemas [12]. Debido a que es una rama del conocimiento en permanente construcción y desarrollo, diferentes autores plantean sus ontologías en relación con el mantenimiento del software. A continuación, se describe algunos de los trabajos más representativos.

2.1 La ontología informal para el mantenimiento del software [21]

Describe los aspectos más importantes que se deben tener en cuenta para realizar estudios empíricos del mantenimiento del software. Aunque la propuesta tiene como objetivo particular identificar los factores de dominio que influyen en los resultados de estos estudios, propone otros que influyen en la fase y que sirven como para modelar una ontología de mantenimiento. La propuesta está estructurada en cuatro sub-ontologías: 1) de productos, en la que se define los productos software que se van a mantener, así como su estructura interna, composición y versiones existentes; 2) de actividades, en la que se define las actividades y recursos, dos tipos de elementos básicos en la gestión de un proyecto de mantenimiento; 3) de procesos de la organización, en la que se define cómo llevar a cabo las actividades y cómo organizar el proceso de mantenimiento como tal; y 4) de agentes, que abarca la jerarquía de tipos de agentes existentes en la gestión de los proyectos de mantenimiento.

Aunque es una propuesta que sirve como referencia para otras ontologías, algunas de las cuales se comentan en este trabajo, no llega a ser un modelo que direccione una aplicación de ontología de mantenimiento de software, ni tiene en cuenta el concepto de gestión del conocimiento en un contexto en el que sea posible aprovecharlo eficientemente por parte de los probadores.

2.2 El enfoque orientado al concepto [22]

Esta ontología parte de la definición abstracta de Gruber [20], en la que se acepta como una especificación explícita de una conceptualización, y refuerza la tesis de que representa cierto punto de vista sobre una solicitud de dominio, en el que hay que definir los conceptos que viven en él, de modo explícito y sin ambigüedades. Esta ontología complementa la definición de Gruber al tener en cuenta el papel que desempeñan los conceptos de la especificación explícita. Una ontología debe ser una obra de referencia, por lo que debe existir el cumplimiento de un estricto compromiso ontológico para que los usuarios puedan acceder a los significados de los conceptos.

El objetivo de la propuesta es considerar un enfoque ligero en el que es deseable, pero no necesario, alcanzar una serie de conceptos que se puedan usar como estándar para una comunidad más grande, y no la de crear ontologías formales y rigurosas para aplicar sobre dominios sometidos a consideración. Este objetivo se refuerza al definir que una de las principales actividades en el desarrollo de software es obtener conocimiento y entendimiento acerca del dominio de la aplicación.

Aunque esta propuesta apoya dicha actividad con una amplia gama de herramientas y técnicas, no tiene en cuenta el hecho de que mucho conocimiento continúa implícito en los objetos

resultantes: los vínculos entre los diferentes objetos, el conocimiento que se pierde como resultado de las mejoras iterativas o el conocimiento que las partes implicadas consideran de sentido común. Además, al considerar ontologías que busquen facilitar los procesos del mantenimiento del software, es necesario intentar un acercamiento conceptual unificado para compilar una técnica común, lo que incrementa las posibilidades para nombrar y clasificar cada concepto. Es necesario trabajar en dos frentes: 1) en la posibilidad de especificar cuáles temas pueden ser similares al incluir la definición de relaciones entre temas de conocimiento, y 2) en la posibilidad de agrupar temas en varias categorías.

2.3 La ontología basada en el conocimiento [23]

El principio en el que se basa esta propuesta es que el desarrollo y el mantenimiento de software deben ser tareas de conocimiento intensivo, en las que se necesita conocer el dominio de la aplicación del software, el problema que soluciona el sistema, los requisitos del problema, la arquitectura del sistema, la forma como encajan las diferentes partes entre sí, y cómo el sistema interactúa con el medio ambiente. Casi siempre este conocimiento no se documenta y reside en la memoria de los ingenieros de software, por lo que es inestable: una organización puede pagar varias veces a los profesionales para redescubrir el conocimiento previamente adquirido y posteriormente perdido.

Esta propuesta se estructura en cinco aspectos: toma los cuatro de Kitchenham [21] y le agrega un quinto aspecto, no considerado en la ontología informal: el conocimiento relacionado con el dominio de aplicación, en el que estructura una ontología orientada a determinar el conocimiento que necesitan los encargados de realizar mantenimiento del software. Parte de la actividad y aplicación de esta ontología se desarrolla en posteriores trabajos de los autores [24, 25], en los que refrendan el concepto de desarrollo y mantenimiento de software como la comprensión de las necesidades de los usuarios y su mundo (dominio de aplicación, reglas de negocio), y un proceso en el que se convierte el código del funcionamiento en la aplicación de una serie de decisiones de diseño. Todo esto (dominio de aplicación, normas, decisiones de diseño) representa el conocimiento que se incrusta en la aplicación resultante, muy a menudo de forma no registrada.

No es claro en esta propuesta cómo definir un mapa de conocimiento alrededor del mantenimiento del software, ya que el tiempo que requiere la captura y el análisis de la información se convierte en factor crítico que no se especifica claramente en ella. Por eso es que muchas empresas optan por acercarse a las fábricas de software para realizar este proceso, costos que muchas organizaciones no pueden cubrir o no tienen presupuestado en sus proyectos.

2.4 MANTIS: Entorno para el mantenimiento integral del software [26]

Es un grupo de ontologías orientadas al uso descrito en la propuesta de Gruninger y Lee [27], aunque solo se orienta a dos de los tres usos que proponen: la comunicación (entre los integrantes de los grupos de mantenimiento y entre personas y sistemas), y la reutilización y organización de conocimiento. La inferencia computacional, el tercer uso propuesto en aquella, no se incluye en la actual MANTIS, se propone como una de las líneas de trabajo futuras. Al mantener limitados los usos de las ontologías a los ya descritos, no es necesario que se formalicen completamente; en su lugar, la representación de la información se logra mediante los modelos y metamodelos.

La propuesta presentada en esta tesis no implica solo aspectos de gestión, ya que también aboga por integrar, desde una perspectiva más amplia de proceso de negocio, la dimensión de gestión

con la dimensión de Ingeniería del Software. Para ello se define un *entorno extendido para la gestión de proyectos de mantenimiento de software*, que integra y amplía dos conceptos básicos: metodología y entorno de Ingeniería del Software; define un marco de trabajo conceptual, considerado imprescindible para que las colecciones de herramientas metodológicas y técnicas formen un Sistema de Información automatizado integrado, y útil para la gestión de proyectos de mantenimiento. Este marco de trabajo da el soporte conceptual a todas las herramientas metodológicas y técnicas que describe y, por tanto, desempeña un papel central para cualquier propuesta en el tema.

MANTIS propone un entorno extendido que engloba tres tipos de herramientas: conceptuales, metodológicas y técnicas. Sus principales características son: integración por medio de herramientas, orientación a procesos, especialización en mantenimiento, escalabilidad y adaptabilidad. El componente ontológico de esta propuesta es uno de los más completos de los referenciados en esta investigación, ya que tiene en cuenta la fase de mantenimiento del software como un componente integral del proceso de la Ingeniería del Software y, porque mediante el uso de modelos y metamodelos, se acerca a conceptos como el orientado por objetos y la tendencia a los aspectos.

2.5 La ontología basada en la reutilización de la información [28]

Esta ontología parte de la premisa de que es conveniente, para toda organización, que la información y el conocimiento se procesen y almacenen, de forma tal que se puedan reutilizar y que, para el mantenimiento, es importante realizar una buena gestión de los mismos, ya que proceden de distintas fuentes y etapas del ciclo de vida. Describe la manera de definir los conceptos involucrados en el mantenimiento del software y cómo representarlos en una ontología, potencializando el reúso de la información mediante el uso de técnicas de razonamiento basado en casos, de tal forma que los encargados del mantenimiento aprovechen las experiencias y lecciones aprendidas por otros.

Tiene origen en la ontología de Kitchenham et al. [21] al identificar los factores que influyen en el mantenimiento, pero, además, indica un conjunto de aspectos dinámicos del dominio descritos en términos de estados, eventos y procesos; es decir, esta ontología se extiende mediante la especificación de aspectos estáticos y dinámicos desglosados en tres ontologías y cuatro sub-ontologías. Su objetivo es definir una ontología con un nivel de abstracción más complejo y, aunque siempre se enfoca en la gestión de proyectos de mantenimiento de software con un punto de vista de procesos del negocio, pierde de vista el concepto dinámico del producto de la Ingeniería del Software: el sistema. No tiene en cuenta que los sistemas no son estáticos, que evolucionan de la misma forma que lo hacen los procesos del negocio, lo que debilita el concepto al momento de realizar el mantenimiento como una actividad inmersa en alguno de ellos.

Aunque la idea general es que se pueda utilizar los conocimientos alcanzados en proyectos anteriores, como ayuda para realizar los futuros, no se encuentra una clara y exhaustiva definición de cómo hacer que la reutilización del conocimiento sea útil en la realización del mantenimiento del software. Es un área cuyos aportes están cortos y se hace necesario comenzar trabajos de soporte que, a corto plazo, permitan la reutilización del conocimiento.

2.6 La ontología de conceptos de ingeniería de software [29]

Representa la idea de crear una ontología para separar los conocimientos de Ingeniería del Software del dominio de los conocimientos sobre las operaciones, componentes de software y

sistema de metadatos. Además, permite hacer supuestos explícitos sobre el dominio, con lo que es posible hallar lagunas en el conocimiento acerca de la forma como se estructura algunos lenguajes de programación orientados por objetos [30].

Esta ontología describe la relación entre los componentes de un software orientado por objetos (programas que contienen los paquetes de clases, las clases abstractas y las interfaces con sus métodos) y las pruebas de software, las métricas, los requisitos y sus relaciones, definidos como los distintos componentes software. Las métricas y las pruebas se asocian con un componente software y tienen momentos en que se calcula su valor varias veces. Los requisitos se asocian con múltiples componentes y pueden codificarse con una o más clases orientadas por objetos; un método se puede designar como el punto de entrada para la exigencia, y un punto de acceso proporciona una idea por dónde empezar a rastrear la aplicación en el código fuente.

El momento clave para utilizar esta ontología es al hacer la captura de información de los cambios, ya que se realiza mediante un objeto de propiedad que se puede utilizar en cualquier componente software, en pruebas, métricas o requisitos, y denota cuándo se modificó por última vez. Además, el documento describe la aplicación del componente, que se refiere a los requisitos, para explicar las decisiones generales del diseño. Es de esperar que este tipo de ontologías, con la codificación de características comunes del dominio de la Ingeniería del Software, tenga un alto componente de reutilización.

2.7 La ontología basada en el conocimiento del sistema [31]

Se basa en la ontología orientada a la gestión de proyectos de mantenimiento del software [19], en la que se busca representar los aspectos estáticos y dinámicos del proceso de mantenimiento desde el punto de vista de los procesos de negocio. Establece una relación entre la ontología y el conjunto de mejores prácticas, contenidas en la integración de modelos de capacidad y madurez de CMMI, según la cual, es difícil especificar la transferencia de conocimientos mediante las buenas prácticas que describen los modelos de madurez. Esto lo ratifica al explicar el proceso mediante el cual se forma a un asesor como nuevo integrante en un proceso para mejorar una actividad. Asimismo, es difícil referirse a la rapidez o al acceso a una práctica, o sub-conjunto, cuando se intenta responder preguntas durante o después de la evaluación de madurez.

Aunque esta propuesta describe el proyecto de modelado de un software de mantenimiento con base en la metodología de van Heijst [32], en la que se representa el conocimiento de sentido común como reutilizable a través de dominios, su ontología resultante la conforman: el modelo de tareas sobre el concepto de un formalismo de la ontología de dominio, la cartografía de la ontología en el papel del conocimiento de la tarea y un modelo resultante al instanciar la aplicación del dominio específico del conocimiento. Identificar las mejores prácticas en un modelo de madurez es una tarea difícil, debido al elevado número de conceptos y las múltiples respuestas adecuadas relacionadas con cada uno de ellos, por lo que es necesario pensar en una ontología que ayude a encontrar esas mejores prácticas para la fase de mantenimiento.

Esta ontología no tiene en cuenta muchos conceptos que participan en la solución, ya que comienza con un número limitado debido a que su propósito es mostrar la utilidad de una ontología en el mantenimiento del software de dominio. Los modelos de madurez suelen incluir detalles de las mejores prácticas que podrían ayudar en la solución de este tipo de problemas, pero la cuestión es que las mejores prácticas y sus interrelaciones permanecen ocultas en la arquitectura del modelo de madurez, especialmente en el proceso de dominio y planes de trabajo. Debido a esto es necesario encontrar una forma de vincular esta arquitectura con una ontología

de los conceptos de mantenimiento, además de analizar las tareas necesarias para construir un sistema de Gestión de Conocimiento que, quienes realizan la actividad, puedan utilizar de apoyo en su búsqueda de soluciones [33].

3. PROPUESTA METODOLÓGICA PARA EL DISEÑO DE UNA ONTOLOGÍA

Una taxonomía se debe pensar como un criterio que determina una jerarquía resultante, que se refleja en el resto de relaciones que define y dota de expresividad al conjunto; por lo que crear una ontología implica realizar elecciones, seleccionar criterios y, dado que su objetivo es servir de referencia a personas y aplicaciones, se requiere que esas elecciones y categorizaciones estén consensuadas y reconocidas por éstas. Una ontología es la descripción conceptual y terminológica de un conocimiento compartido acerca de un dominio específico y, aparte de la formalización e interoperabilidad de aplicaciones, no es más que la principal competencia del término: hacer mejoras en la comunicación utilizando un mismo sistema en lo terminológico y conceptual [34].

La metodología que se propone en esta investigación se basa en la propuesta de Noy y McGuinness [30], quienes definen todos los conceptos relevantes acerca de por qué desarrollar una ontología, y presentan una metodología para su diseño basada en los sistemas de representación del conocimiento declarativo. La experiencia de los autores en la construcción y mantenimiento de ontologías en un variado número de ambientes, incluyendo Protege-2000, Ontolingua y Chimaera, se refleja en el contenido del texto. Y, aunque no se encuentra una única forma ni metodología *correcta* para estructurar y desarrollar ontologías, a continuación, se describe los puntos más generales que se debe considerar, además de los procedimientos posibles para diseñarlas. Es una forma iterativa de desarrollo que comienza por abordar el objetivo de forma frontal, para luego afinarlo y complementarlo con detalles y metas alcanzadas.

Este proceso metodológico se enmarca en algunas reglas que, aunque dogmáticas, ayudan a tomar decisiones en el diseño ontológico:

1. El proceso de modelar un dominio no tiene una única forma correcta de hacerlo, siempre existe otras alternativas; la selección siempre depende de la idea que se tiene para aplicarla y de las futuras extensiones que se visualicen.
2. El proceso de diseñar y desarrollar una ontología es necesariamente iterativo.
3. Cada concepto a incluir en la ontología debe considerarse desde la descripción de los objetos físicos o lógicos involucrados en ella, y desde las relaciones que tenga el domino seleccionado, es decir los sustantivos (objetos) o verbos (relaciones), en las oraciones que lo describen.
4. También se debe tener en cuenta que la ontología modela la realidad del mundo, por lo que los conceptos en ella deben reflejarla.
5. Luego de tener los primeros acercamientos al diseño de la ontología es necesario revaluarlos y discutirlos con especialista en el área; este proceso iterativo es necesario durante el transcurso del ciclo de vida de la ontología en construcción.

A continuación, se detalla los pasos de la metodología propuesta:

3.1 Determinar el dominio y el alcance

El primer paso metodológico en esta propuesta es determinar el dominio, que se puede lograr al realizar preguntas como: ¿Qué dominio cubrirá la ontología? ¿Qué uso se le dará? ¿A qué tipo de

preguntas deberá responder la información en ella? ¿Quién se encargará de usarla y de mantenerla? Las respuestas a estas preguntas normalmente cambian en el proceso de estructuración de la ontología, pero la información que se recolecta sirve de ayuda para no exceder los límites y alcances del diseño.

Para determinar el alcance se utiliza preguntas de competencia que, como bosquejo, no requieren ser exhaustivas, además, la base de conocimientos de la ontología debería responderlas [35], y posteriormente se podrán utilizar para realizar las pruebas de control de calidad: ¿Contiene la ontología información suficiente para responder ese tipo de preguntas? ¿Qué nivel de detalle o representación de un área en particular requieren las respuestas?

3.2 Considerar la reutilización de otras ontologías

Muchas veces vale la pena considerar otros trabajados acerca de la temática para verificar si es posible utilizarlos para refinar y extender la ontología que se diseña [36]. El reúso de ontologías puede hacerse cuando la que se estructura requiere interactuar con otras aplicaciones, ya que puede contar con dominios claros y específicos; o cuando el nivel de formalismo en el que otras se expresan pasa a segundo plano, ya que la mayoría de sistemas de representación de conocimiento tiene la posibilidad de importarlas y exportarlas; además, la labor de *traducir* una ontología desde un formalismo particular a otro, no es una tarea complicada [37, 38].

En la Internet y en la literatura existe bibliotecas de ontologías que se pueden reutilizar:

- Biblioteca de Ontolingua: http://www.ksl.stanford.edu/software/ontolingua/
- Guía de recursos sobre ontologías: http://es.geocities.com/ontologias_y_tesauros/guia_de_recursos_sobre_ontologias.htm
- La biblioteca de ontologías DAML: http://www.daml.org/ontologies/
- Biblioteca semántica WebQuest: http://cfievalladolid2.net/webquest/common/index.php

Igualmente, es posible adquirir ontologías comerciales:

- UNSPSC: www.unspsc.org
- RosettaNet: www.rosettanet.org
- DMOZ: www.dmoz.org

En cualquier caso, sea porque se estructura desde cero o porque se reutiliza otra, la conceptualización y elaboración de una ontología se debe realizar para cada una de las ontologías parciales definidas en el alcance teniendo en cuenta [39]:

1. Definir el glosario de conceptos a partir de las fuentes de conocimiento citadas.
2. Definir las interrelaciones semánticas entre dichos conceptos, representándolas mediante un diagrama de clases UML, y elaborar una tabla de clases de interrelaciones con las diversas clases identificadas.
3. Analizar los conceptos relacionados para identificar las partes comunes a dos o más conceptos, y decidir si estas partes son a su vez conceptos y, en su caso, incluirlos en el glosario de conceptos.
4. Identificar los atributos terminales (atributos normales) de todos los conceptos, e incorporarlos en las tablas de atributos y el diagrama de clases UML.
5. Completar las tablas de atributos de conceptos e incluir los atributos no terminales.

6. Comprobar la completitud de todas las tablas de atributos e indicar si pertenece a la capa de descripción del artefacto, de su interfaz o contexto.

3.3 Enumerar términos importantes para la ontología

Para cualquier forma de trabajo ontológico es muy útil realizar una lista que contenga todos los términos con los que se va a hacer enunciados o para explicar al usuario, por lo que es necesario, como método inicial de trabajo, conocer cuáles con los términos que se tratarán en la ontología y qué propiedades tienen. Se debe estructurar una lista que integre estos términos, pero sin tener en cuenta los conceptos que representan, las relaciones entre ellos, las propiedades que puedan tener o si los conceptos son clases o *slots*, ya que, en esta metodología, se tienen en cuenta en las siguientes fases [40]. De lo que se trata es de crear las definiciones de los conceptos en la jerarquía para luego describir sus propiedades de forma sucesiva.

3.4 Definir las clases y la jerarquía de clases

Existe varios enfoques para desarrollar jerarquías de clases [41]:

- El enfoque *top-down*, en el que primero se define los conceptos más generales en el dominio y su respectiva especialización.
- El enfoque *bottom-up*, que comienza con la definición de clases específicas, para luego generar las hojas de la jerarquía con su respectivo agrupamiento de clases en conceptos más generales.
- El enfoque de desarrollo combinado, que resulta de combinar los enfoques anteriores y que comienza por definir los conceptos más importantes, para luego generalizarlos y especializarlos apropiadamente.

Ninguno de los tres enfoques puede considerarse mejor a los otros, por lo que decidir alguno depende en gran medida de la visión que se tiene del dominio: si se tiene una visión sistemática *top-down* lo lógico es utilizar el enfoque *top-down*; pero el enfoque combinado es en muchas ocasiones el más fácil de aplicar, ya que los *conceptos del medio* generalmente son los más descriptivos en el dominio [42]. Si la tendencia de la estructura es pensar primero en una clasificación más general, entonces podría funcionar mejor el enfoque *top-down* y si se prefiere comenzar con un listado de ejemplos específicos, el enfoque *bottom-up* podría ser el más apropiado.

Sea cual sea el enfoque que se elija la primera tarea es definir las clases: de la lista de términos que se creó en el paso 3 se selecciona los que describen objetos con existencia independiente; estos términos se convierten en las clases de la ontología y son la base de la jerarquía de clases. También es posible ver las clases como preguntas que tienen un argumento (predicados unarios). Luego se organizan las clases en una taxonomía jerárquica, teniendo en cuenta que una instancia de una clase puede ser instancia de otra: s*i una clase X es una superclase de la clase Y, entonces cada instancia de Y lo es también de X.* Es decir, que la clase *B* representa un concepto que es un *tipo de A* [43].

3.5 Definir las propiedades de las clases

Las clases aisladas no ofrecen la suficiente información para responder a las preguntas de competencia del paso 1, por lo que, luego de definir las clases es necesario describir la estructura interna de conceptos: se selecciona clases de la lista de términos estructurada en el paso 3 y, muy

probablemente, los términos restantes son las propiedades de ellas, es decir que la describen; para cada propiedad en la lista se determina a qué clase describe, con lo que se convierten en los *slots* de la clase [20].

Existe varios tipos de propiedades que se pueden convertir en *slots* en una ontología:

- Las intrínsecas: sabor, color, tamaño.
- Las extrínsecas: nombre, área origen.
- Partes: si el objeto es estructurado pueden ser partes físicas o abstractas: platos de una comida, vino que acompaña.
- Relaciones con otros individuos: relaciones entre los miembros individuales de una clase y otros ítems: casa productora, que relaciona un producto con un fabricante y con la materia prima con la que se fabrica.

Además de las propiedades identificadas previamente, es necesario añadir los *slots* a la clase. Se debe tener en cuenta que todas las sub-clases de una clase heredan los *slots* de la misma, y que un *slot* debe estar yuxtapuesto a la clase más general que puede tener esa propiedad [42].

3.6 Definir las facetas de los slots

Los *slots* tienen diferentes facetas para describir el tipo de valor [42]:

- *Cardinalidad*. Define cuántos valores puede tener un *slot*. Existe sistemas que solo aceptan cardinalidad simple (solo un valor) o cardinalidad múltiple (cualquier cantidad de valores); otros admiten la especificación de cardinalidades mínimas y máximas para describir con mayor precisión la cantidad de valores del *slot*.

- *Tipo de valor*. Describe qué tipos de valores puede contener el *slot*: *String*, el valor es una cadena de caracteres; *Number,* describe *slots* con valores numéricos; *Boolean,* son valores de bandera si/no, verdadero/falso; *Enumerated,* denotan una lista específica de valores admitidos para el *slot*, los define el desarrollador; *Instance,* admiten la definición de relaciones entre individuos; algunos sistemas especifican solo el tipo de valor con la clase, en lugar de un enunciado especial de *slots* de tipo *instance*.

- *Dominio y rango*. Comúnmente, las clases admitidas para los *slots* tipo *Instance* se nombran como rango de éste; las clases a las cuales un *slot* está yuxtapuesto, o las clases de las que un *slot* describe sus propiedades, se nombran como dominio. Las reglas para determinar dominio y rango de un *slot* son: al definir un dominio o rango de un slot se debe encontrar la o las clases más generales, que puedan ser el dominio o rango de los slots; no se debe definir un dominio o rango muy general: todas las clases en el dominio deben ser descritas por el *slot*, y las instancias de las clases en el rango de un *slot* deben ser rellenos potenciales del mismo [41].

3.7 Crear instancias y cardinalidades

El último paso de esta metodología es crear las instancias individuales de clases en la jerarquía: 1) elegir la clase, 2) crear la instancia individual de la clase, y 3) llenar los valores del slot [44, 45]. Debe tenerse en cuenta: 1) decidir si un concepto particular será una clase o una instancia individual en la ontología, y que depende de las aplicaciones que tendrá [46, 47]; 2) para decidir

dónde terminan las clases y comienzan las instancias es necesario hallar primero el nivel más bajo de granularidad en la representación, y que también lo determina la aplicación potencial de la ontología [48]; 3) solo las clases se pueden representar en una jerarquía, ya que los sistemas de representación de conocimiento no tienen la noción de sub-instancias, por lo que, si los términos tienen una jerarquía natural, se deben definir como clases aunque no tengan instancias [49, 50].

4. CONCLUSIONES

Debido a que la Gestión del Conocimiento es una técnica importante para facilitar el trabajo de los encargados del mantenimiento del software, es necesario contar con un método para detectar las fuentes existentes y su ubicación en la organización; una manera de hacerlo, rápida y eficientemente es utilizar agentes inteligentes mediante ontologías, para detectar la información que sea útil y acorde con las necesidades de la tarea de mantenimiento.

En este documento se presenta el análisis de la conceptualización alrededor de la Gestión del Conocimiento mediante ontologías para el mantenimiento del software, además, se describe una metodología para desarrollarla; sin embargo, luego de seguir y aplicar los pasos, reglas y recomendaciones, hay que recordar que no existe una sola ontología que se pueda considerar correcta para un dominio dado.

El diseño ontológico es un proceso que exige creatividad, por lo que las ontologías nunca serán iguales, aunque se estructuren sobre el mismo dominio. La aplicación potencial de la ontología, así como la comprensión y aspecto del dominio por parte del diseñador, afectan las opciones del diseño mismo de la ontología.

No se sabe si algo es bueno hasta que se le pone a prueba, por lo que se puede evaluar la calidad de la ontología propuesta solamente utilizándola en las aplicaciones para las cuales se diseñó.

Para probar la metodología propuesta se debe plantear la posibilidad de crear una ontología en una herramienta automatizada, y extenderla con algún lenguaje de definición de reglas, para superar las limitaciones del lenguaje natural en cuanto a la composición de los escenarios. Se recomienda elegir uno que se ajuste a las necesidades de la implementación, ya que el objetivo final es encontrar la manera de definir reglas que faciliten el entendimiento entre los sistemas y las personas.

Otro trabajo futuro puede estar relacionado con mejorar los procesos de Gestión del Conocimiento. En esta propuesta se descarta el conocimiento inconsistente al mantenimiento del software, y debido a que la misma ontología puede ser parcialmente inconsistente, podría mejorarse si fuera posible descartar únicamente las partes inconsistentes, en vez de toda la ontología. Para lograrlo, puede ser que al modificar las ontologías fuente y eliminar de ellas el contenido inconsistente se logre una ontología final más consistente.

El marco formal que se presenta en este trabajo para integrar ontologías se sitúa en el contexto de Ingeniería del Software y la Gestión del Conocimiento, dado que los resultados que se obtuvieron son significantes para las dos áreas de investigación. Para la primera, porque proporciona un modelo y un marco formales para desarrollar proyectos en la fase de mantenimiento del software a través de ontologías; para la segunda, porque la metodología presentada contribuye a mejorar la Gestión del Conocimiento y el desarrollo de sistemas para el mantenimiento y, particularmente, la aplicación de la metodología presentada en este trabajo ayuda a simplificar la mayoría de aspectos relacionados con el conocimiento en el área.

REFERENCIAS

[1] Lindvallv I. (2002). Knowledge management in software engineering. IEEE Software19(3), 26–38.

[2] Pigoski T. (1996). Practical software maintenance: Best practices for managing your software investment. John Wiley.

[3] Walz D. et al. (1993). Inside a software design team: Knowledge acquisition, sharing, and integration. Communications of the ACM 36(10), 63–77.

[4] Rodríguez O. et al. (2004). Understanding and supporting knowledge flows in a community of software developers. Lecture Notes in Computer Science 3198, 52–66.

[5] Dingsøyr T. y Conradi R. (2002). A survey of case studies of the use of knowledge management in software engineering. Inter. Journal of Soft. Engineering and Knowledge Engineering 12(4), 391–414.

[6] Nebus J. (2001). Framing the knowledge search problem: Whom do we contact, and why do we contact them? En Academy of Management Best Papers Proceedings. New York, USA.

[7] Mamone S. (1994). The IEEE standard for software maintenance. ACM SIGSOFT Software Engineering Notes 19(1), 75–76.

[8] Singh R. (1994). ISO/IEC draft international standard 12207, software life-cycle processes. IFIP Transactions A-55, 111–119.

[9] Pressmann R. (2014). Software engineering: A practitioner's approach. McGraw-Hill.

[10] IEEE. (1998). IEEE Std 1061-1998. IEEE standard for a software quality metrics methodology. Technical report. IEEE Press.

[11] Lefort L. et al. (2006). Towards scalable ontology engineering patterns: Lessons learned from an experiment based on W3C's part-whole guide-lines. En second Australasian workshop on advances in ontologies. Sidney, Australia.

[12] Horrocks I. (2008). Ontologies and the semantic web. Communications of the ACM 51(12), 58–67.

[13] Evans J. (2008). Electronic publication and the narrowing of science and scholarship. Science 321(5887), 395–399.

[14] Lacy L. (2005). OWL: Representing information using the Web ontology language. Trafford Publishing.

[15] Gruber T. (1995). Towards principles for the design of ontologies used for knowledge sharing. International Journal of Human–Computer Studies 43(5–6), 907–928.

[16] Reuver M. y Haaker T. (2009). Designing viable business models for context-aware mobile services. Telematics and Informatics 26(3), 240–248.

[17] Mahesh K. (1996). Ontology development for machine translation: Ideology and methodology. Technical report MCCS-96-292.New Mexico State University.

[18] Oliveira K. et al. (2003). Knowledge for software maintenance. En Fifteenth international conference on software engineering and knowledge engineering. San Francisco, USA.

[19] Ruiz F. et al. (2004). An ontology for the management of software maintenance projects. International Journal of Software Engineering and Knowledge Engineering 14(3), 323–349.

[20] Gruber T. (1993). A translation approach to portable ontology specifications. Knowledge Acquisition 5(2), 192–220.

[21] Kitchenham B. et al. (1999). Towards an ontology of software maintenance. Journal of Software Maintenance: Research and Practice 11(6), 365–389.

[22] Deridder D. (2002). Facilitating software maintenance and reuse activities with aconcept-oriented approach. Technical report. Vrije Universiteit Brussel.

[23] Oliveira K. et al. (2003). Organizing the knowl-edge used in software maintenance. Journal of Universal Computer Science 9(7),641–658.

[24] Oliveira K. et al. (2004). Legacy software evaluation model for out sourced maintainer. En eighth European conference on Software maintenance and reengineering. Madrid, Spain.

[25] Oliveira K. et al. (2007). Software maintenance seen as a knowledge management issue. Information and Software Technology 49(5), 515–529.

[26] Ruiz F. (2003). MANTIS: Entorno para el Mantenimiento Integral del Software. Disertación doctoral. Universidad de Castilla.

[27] Gruninger M. y Lee J. (2002). Ontology applications and design. Comm. of the ACM 45(2), 39–41.

[28] Vizcaíno A. et al. (2006). Aplicando gestión del conocimiento en el proceso de mantenimiento del software. Revista Iberoamericana de Inteligencia Artificial 10(31), 91–98.

[29] Hyland D. et al. (2006). Enhancing software maintenance by using semantic web techniques. En 5th international semantic web conference. Londres, UK.
[30] Noy N. y McGuinness D. (2001). Ontology development 101: A guide to creating your first ontology. Stanford Medical Informatics Technical Report SMI-2001-2800.
[31] April A. et al. (2006). A formalism of ontology to supporta software maintenance knowledge-based system. En eigh-teenth international conference on software engineering & knowledge engineering conference. San Francisco, USA.
[32] van Heijst G. et al. (1996). Using explicit ontologies in KBS development. International Journal of Human–Computer Studies 46, 2–3.
[33] Sarder B. y Ferreira S. (2007). Developing systems engineering ontologies. En IEEE international conference of system of systems engineering. San Diego, USA.
[34] Park J. et al. (2008). Product ontology construction from engineering documents. En International conference on smart manufacturing application. El Cairo, Egypt.
[35] Gruninger M. y Fox M. (1995). Methodology for the design and evaluation of ontologies. En workshop on basic ontological issues in knowledgesharing. Montreal, Canada.
[36] Gómez A. (1998). Knowledge sharing and reuse. In J. Liebowitz (Ed.), The handbookof applied expert systems pp. 45-68). CRC Press.
[37] Musen M. (1992). Dimensions of knowledge sharing and reuse. Computers and Biomedical 25(5), 435–467.
[38] Rothenfluh T. et al. (1996). Reusable ontologies, knowledge-acquisition tools, and performance systems: PROT'EG'E-II solutions to Sisyphus-2. International Journal of Human–Computer Studies 44(3–4), 303–332.
[39] Tautz C. y von Wangenheim C. (1999). REFSENO: A representation formalism for software engineering ontologies. Technical Report IESE-Report No. 015.98/E. Fraunhofer Institute for Experimental Software Engineering.
[40] Borgida A. et al. (1996). CLASSIC: A structuraldata model for objects. En ACM SIGMOD international conference on management of data. Portland, USA.
[41] Uschold M. y Gruninger M. (1996). Ontologies: Principles, methods and applications. Knowledge Engineering Review 11(2), 93–155.
[42] Rosch E. (1978). Principles of categorization. En E. Rosch y B. Lloyd (Eds.), Cognition and categorization (pp. 123-146). MIT Press.
[43] Ning H. y Shihan D. (2006). Structure-based ontology evaluation. En IEEE international conference business engineering. Los Angeles, USA.
[44] McGuinness D. y Wright J. (1998). An industrial strength description logic-based configurator platform. IEEE Intelligent Systems 13(4), 69–77.
[45] Choi K. (2007). IT ontology and semantic technology. En International conference on natural language processing and knowledge engineering. Estambul, Turky.
[46] Fernández M. et al. (2007). Methontology: From ontological art towards ontological engineering. En AAAI Spring Symposium. University of Stanford, USA.
[47] García R. y Piattini M. (2003). Calidad en el desarrollo y mantenimiento del software. Editorial Rama.
[48] IEEE. (1995). STD 1074-1995: IEEE standard for developing software life cycle processes. Research 25(1), 435–467.
[49] Ruiz F. et al. (2006). Ontologies for software engineering and software technology. Springer.
[50] Zhang L. et al. (2008). User defined ontology change and its optimization. En Control and decision conference. Beijin, China.

CAPÍTULO XXII
Desafíos y estrategias prácticas de los estudios empíricos sobre las técnicas de prueba del software[1]

Edgar Serna M.[1]
Fernando Arango I.[2]
[1]Instituto Antioqueño de Investigación
[2]Universidad Nacional de Colombia

En este capítulo se discute una serie de cuestiones que típicamente se plantean cuando se realizan estudios empíricos con técnicas de prueba del software. Aunque algunos problemas son generales a todas las disciplinas empíricas, los estudios de prueba del software enfrentan una serie de desafíos específicos. Algunos de los más importantes se discuten a continuación.

[1] Publicado en la Revista Ingeniería y Competitividad 13(1), 141-146. 2011.

INTRODUCCIÓN

La prueba es una importante actividad de ingeniería, y es responsable de una porción significativa de los costos del desarrollo y el mantenimiento del software [1, 2], por lo que es necesario que investigadores y profesionales comprendan sus ventajas y desventajas, lo mismo que los factores que influyen en las técnicas que utilizan para llevarla a cabo. Se puede obtener algunos conocimientos mediante el uso de marcos analíticos, relaciones de subsunción o por axiomas [3-5]; sin embargo, en general, las técnicas de prueba son heurísticas y su desempeño varía de acuerdo con los diferentes escenarios, por lo que deben ser estudiadas empíricamente.

Los estudios empíricos de las técnicas de prueba, como los estudios de los ingenieros que realizan las pruebas, implican muchos desafíos y ventajas y desventajas de costo-beneficio, lo que limita el progreso de los estudios en esta área. En general, se puede considerar dos clases de estudios empíricos de pruebas del software: 1) los experimentos controlados, y 2) los estudios de caso. Los experimentos controlados se centran en un riguroso control de las variables, en un intento por preservar la validez interna y apoyar conclusiones acerca de la causalidad, pero las limitaciones que resultan de ejercer este control pueden restringir la capacidad de generalizar los resultados [6]. Por su parte, los estudios de caso sacrifican el control, por lo tanto, la validez interna, pero pueden incluir un contexto más amplio [7]. Cada una de estas clases de estudios puede ofrecer una perspectiva sobre las técnicas de prueba del software, y juntos son complementarios.

Sin importar el tipo de estudio aplicado, en este capítulo se analiza una serie de cuestiones que se plantean típicamente cuando se realizan estudios empíricos acerca de las técnicas de prueba del software. Aunque algunos problemas son generales a todas las disciplinas empíricas, los estudios de prueba del software enfrentan una serie de desafíos específicos, y algunos de los más importantes se discuten en este trabajo.

1. DESAFÍOS ESPECÍFICOS DE LA PRUEBA DEL SOFTWARE

1.1 Siembra de fallas

Uno de los principales temas que se debe enfrentar cuando se evalúa y compara técnicas de prueba es la cuantificación de su eficacia en la detección de fallas. Idealmente, las fallas en el software deberían representar las fallas de la vida real [8], de modo que los resultados de su eficacia fueran en sí mismos representativos.

Pero existe una serie de problemas con esta estrategia: 1) muchos componentes del mundo real, los sub-sistemas e incluso los sistemas, no tienen un conjunto lo suficientemente grande de defectos que sea adecuado para un análisis cuantitativo. Es necesario recordar que, aunque se espera que las técnicas de prueba estén asociadas con tasas de detección de fallas, esta relación es de naturaleza estadística, por lo que su detección será, hasta cierto punto, un proceso estocástico. Por lo que, muy a menudo, cuando se comparan las técnicas de prueba se están comparando las distribuciones de la detección de fallas, y se recurre a las pruebas de inferencia estadística para determinar si los resultados podrían haberse obtenidos al azar. Para ello es necesario trabajar con muestras de fallas suficientemente grandes, que raramente están disponibles en los sistemas del mundo real. 2) Cuando el referente son las fallas del mundo real, muy probablemente se refieren a una población estadística que no existe, o que cada empresa y sistema estén asociados con diferentes poblaciones de fallas. En otras palabras, si el objetivo es la *falla real*, esa meta probablemente sea difícil de alcanzar. Aunque esto no quiere decir que los estudios industriales no sean útiles.

Debido a lo anterior muchos investigadores tienen que recurrir a la siembra de fallas para llevar a cabo estudios empíricos [9-13], aunque son bien conocidos los problemas asociados a dicha siembra. ¿Cómo ser imparcial y objetivo cuando se siembra fallas con el fin de evaluar una técnica? ¿Cómo garantizar que los resultados obtenidos se pueden generalizar a la población de fallas real? No existe una respuesta adecuada. Sin embargo, los operadores de mutación utilizan una técnica para sembrar fallas [12, 14]. No es posible garantizar que las fallas sembradas sean representativas de una población en particular, pero sí es posible asegurar que una amplia variedad de fallas sea insertada sistemáticamente, de manera un tanto imparcial, al azar [15-17], y probablemente esto es lo mejor que se puede hacer en un entorno artificial.

Se ha hecho estudios con la idea de obtener datos únicos de la efectividad en la detección de fallas y en el costo [13], buscando además una mejor comprensión de qué tipo de fallas son más difíciles de detectar para una determinada técnica [18]. Tales estudios deben complementarse con trabajos de campo, como se discute a continuación. Otra cuestión relacionada con la siembra de fallas es que, si muchas de las sembradas son fáciles de detectar, entonces no es posible diferenciar la eficacia de las técnicas alternativas en la detección de fallas; pero sí resultan útiles para comprobar el porcentaje de casos de prueba que encuentran fallas en un programa defectuoso. Las fallas sencillas, así como las que posiblemente no tienen un impacto sobre el comportamiento del sistema, probablemente se deban ignorar a la hora de evaluar la eficacia.

1.2 Contexto académico vs industrial

Se han dado diversas discusiones acerca del valor de los experimentos en el ambiente académico [19], especialmente sobre la preocupación por el hecho de que las herramientas utilizadas pueden no ser representativas, en términos de escala y complejidad, de sus homólogas industriales; y que cuando participan estudiantes sería natural preguntarse si los resultados obtenidos pueden ser comparados con los de los profesionales, quienes muchas veces tienen más experiencia. Y, por supuesto, como se mencionó antes, que existe un conjunto de fallas con las que se tiene que trabajar y que puede no ser representativo de las fallas detectadas en el campo.

Todas son preocupaciones válidas, pero lo que la experiencia ha demostrado, en el ámbito más general de la Ingeniería del Software empírica, es que no existen estudios empíricos perfectos. Los estudios de campo, en entornos industriales, también implican una serie de problemas:

1. El número de fallas detectadas puede no ser lo suficientemente amplio como para permitir un análisis estadístico cuantitativo.
2. Cuando se trabaja con profesionales activos, a veces no se preocupan por actualizar su campo de formación. En situaciones en las que se requiere evaluar nuevas o avanzadas técnicas, pueden no estar actualizados, especialmente al evaluar el límite de los beneficios potenciales de una técnica cuando se aplica correctamente.
3. Los estudios en el ámbito académico suelen ser más fáciles de controlar. Uno de los retos importantes de la experimentación es asegurarse de que no exista efectos de confusión entre los factores que se estudia (técnicas de la prueba) y otros factores (extraños, humanos [20]. Esto suele ser muy difícil de asegurar en un contexto industrial.

A partir de esta discusión es posible ver que tanto el contexto académico como el industrial tienen puntos fuertes y débiles. Los estudios en el mundo académico a menudo son fuertes en términos de validez interna, ppor ejemplo, la capacidad para sacar conclusiones adecuadas de los datos; y puntos débiles en cuanto a la validez externa se refiere, por ejemplo, es difícil conocer hasta qué punto se puede generalizar los resultados en los contextos industriales. Los estudios de campo

tienen exactamente los puntos fuertes y débiles opuestos. Aunque también se podría argumentar que los resultados en un proyecto determinado en una organización podrían no ser generalizables a otras organizaciones y proyectos.

Se concluye entonces que ambos estudios, el académico y el de campo, son necesarios. Los primeros son útiles para obtener datos únicos de la eficacia de las técnicas y para comprender mejor sus fortalezas y debilidades. Los estudios de campo son más adecuados para evaluar las dificultades al aplicar las técnicas en la práctica y para confirmar los resultados obtenidos en un conjunto real de fallas. También son muy útiles para intentar análisis de costo-eficacia, como el costo de los datos recogidos. Ningún estudio llegará a ser perfecto, o a estar cerca de serlo, pero con el tiempo la acumulación de conocimientos obtenidos en estudios posteriores puede permitir la construcción de un cuerpo de conocimiento. Por último, los estudios en un entorno académico son a menudo el primer paso antes de los estudios en entornos industriales [19].

1.3 Replicación

Lo anterior implica que los estudios se replicarán en diferentes entornos, pero un solo estudio es probable que no tenga un impacto significativo, por lo que es necesario que se replique para que los resultados sean creíbles. La replicación se ha aplicado en otras disciplinas a lo largo del tiempo, y existen técnicas, llamadas meta-análisis, para obtener conclusiones de un conjunto de experimentos [20]. Sin embargo, replicación necesariamente no significa repetición exacta de un estudio. Esto se debe a que los artefactos pueden ser diferentes; la formación y antecedentes de los participantes, si los hay, puede variar considerablemente; incluso el diseño del estudio puede cambiar para hacerle frente a una amenaza en particular por validar; o simplemente el número de observaciones puede ser mayor o menor.

Todo esto afecta la capacidad de obtener efectos visibles. Esta información se debe reportar al escribir de una replicación, de modo que a largo plazo las diferencias entre los estudios se puedan explicar, posiblemente a través de meta-análisis. También sería útil para la comunidad en pruebas del software definir una plantilla de información para reportar cuándo se realizan, y para documentar los experimentos realizados.

1.4 Participación de seres humanos

Los estudios en pruebas del software se pueden clasificar en dos categorías: 1) estudios que involucran sujetos humanos aplicando las técnicas de prueba [18], y 2) las simulaciones, donde las técnicas se aplican mediante simulación de la construcción del plan de pruebas y su ejecución en la versión defectuosa del programa [13].

Ambos tipos de estudios tienen ventajas e inconvenientes. El primero permite evaluar no solo la rentabilidad, sino también la aplicabilidad de las técnicas por ingenieros especializados, lo que explica los factores humanos en las conclusiones; después de todo, los seres humanos son un factor que todavía no puede estar por fuera de las pruebas del software, y es una cuestión importante relacionada con la naturaleza estadística de las tasas de detección de fallas. Para comparar realmente las técnicas de prueba, de forma rigurosa y cuantitativa, se necesita generar un gran conjunto de pruebas para cada una de las técnicas. Esto suele ser inviable cuando los seres humanos están involucrados, y es aquí donde la simulación entra en acción para permitir la generación de un conjunto amplio de pruebas, ejecutarlo en un gran número de programas defectuosos y, por lo tanto, que sean susceptibles de un riguroso análisis estadístico.

Sin embargo, el principal problema es garantizar que el proceso de simulación no introduzca algún sesgo en los resultados, que alguna manera sea de representativo del conjunto de pruebas que pueden generar los seres humanos. Aunque ésta no es la forma como los humanos cometen errores al usar técnicas de prueba, algunas técnicas pueden ser más complejas y dar lugar a más errores, especialmente si no están bien soportadas por herramientas. Así, de nuevo, se concluye que ambos tipos de estudios son necesarios debido a que son complementarios.

2. CONCLUSIONES

Existe una serie de prácticas que pueden ayudar a aliviar los desafíos analizados en este trabajo. Debido al amplio uso de la experimentación, se requiere que esos experimentos se preparen mejor y que exijan más esfuerzo; además, es necesario que los investigadores compartan el material experimental que obtienen de sus aplicaciones. Por otra parte, con la experimentación en un conjunto de sistemas comunes (de referencia) es posible hacer comparaciones más factibles de las técnicas, y ese conjunto de sistemas se puede utilizar como punto de referencia para la validación inicial de las mismas técnicas. Una de las dificultades es que, por lo general, diferentes técnicas de prueba requieren diferentes piezas de información del sistema, por ejemplo, especificaciones en diferentes formas, o tener como objetivo diferentes tipos de sistemas, por lo que elegir puntos de referencia apropiados puede no ser tan fácil.

Los experimentos deben producir una documentación que les permita a otros investigadores replicarlos fácilmente. Idealmente, deberían contener todo el material necesario para realizar el experimento y ser accesible públicamente, bajo ciertas condiciones. Esto le permitiría a la comunidad de investigadores converger mucho más rápido hacia resultados creíbles.

Cada vez que un análisis de datos se realiza en un experimento, sería bueno que dichos datos estuvieran disponibles, incluso si ello implica, de alguna manera, sanearlos cuando la confidencialidad es un problema. Esto permitiría que otro investigador los analizara y, posiblemente, llegara a conclusiones diferentes.

Por último, ya que no es posible esperar que los experimentos sean perfectos en lo que respecta a la validez de todas las fallas, es necesario establecer estándares acerca de cómo llevarlos a cabo y cómo reportarlos. Algunos podrían basarse en otros campos experimentales, pero definitivamente habría que hacer una adecuada adaptación. Esto facilitará la revisión de los trabajos que detallan resultados de pruebas experimentales, y ayudará a la efectiva publicación de resultados más recientes.

REFERENCIAS

[1] Leung H. y White L. (1989). Insights into regression testing. En Confe. on Soft. Mainten. Miami, USA.

[2] Beizer B. (1990). Software Testing Techniques. Van Nostrand Reinhold.

[3] Rapps S. y Weyuker E. (1985). Selecting software test data using data flow information. IEEE Transactions on Software Engineering 11(4), 367–375.

[4] Weyuker E. (1993). More Experience with Data Flow Testing. IEEE Transactions on Software Engineering 19 (9), 912-919.

[5] Rothermel G. y Harrold M. (1996). Analyzing regression test selection techniques. IEEE Transactions on Software Engineering 22 (8), 529–551.

[6] Trochim W. (2000). The Research Methods Knowledge Base. Atomic Dog.

[7] Yin R. (1994). Case Study Research: Design and Methods. Sage Publications.

[8] DeMillo R. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer 11(4), 34-41.

[9] Weyuker E. (1988). The evaluation of program-based software test data adequacy criteria. Communications of ACM 31(6), 668-675.
[10] Hutchins M. et al. (1994). Experiments on the Effectiveness of Dataflow and Controlflow-Based Test Adequacy Criteria. En 16th international conference on Software engineering. Sorrento, Italy.
[11] Graves T. et al. (2001). An Empirical Study of Regression Test Selection Techniques. ACM Transactions on Software Engineering and Methodology 10 (2), 184-208.
[12] Briand L. et al. (2003). Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code. Software - Practice and Experience 33(7), 637-672.
[13] Briand L. et al. (2004). Using Simulation to Empirically Investigate Test Coverage Criteria. En IEEE/ACM International Conference on Software Engineering. Edinburgh, UK.
[14] Offutt A. (1992). Investigations of the Software Testing Coupling Effect. ACM Transactions on Software Engineering and Methodology 1(1), 3-18.
[15] King K. y Offutt A. (1991). A Fortran Language System for Mutation-Based Software Testing. Software-Practice and Experience 21(7), 686-718.
[16] Kim S. et al. (2000). Class Mutation: Mutation Testing for Object-Oriented Programs. En 28th international conference on Software engineering. Erfurt, Germany.
[17] Delamaro M. et al. (2001). Interface Mutation: An Approach for Integration Testing. IEEE Transactions of Software Engineering 27(3), 228-247.
[18] Antoniol G. et al. (2002). A Case Study Using the Round-Trip Strategy for State-Based Class Testing. En 13th IEEE International Symposium on Software Reliability Engineering. Annapolis, USA.
[19] Juristo N. y Moreno A. (2001). Basics of Software Engineering Experimentation. Springer.
[20] Wohlin C. et al. (2000). Experimentation in Software Engineering - An Introduction. Springer.

CAPÍTULO XXIII
Prueba del software: Más que una fase en el ciclo de vida[1]

Edgar Serna M.[1]
Fernando Arango I.[2]
[1]Instituto Antioqueño de Investigación
[2]Universidad Nacional de Colombia

La prueba de software es probablemente la parte menos comprendida del ciclo de vida del desarrollo de software. En este trabajo, mediante una propuesta metodológica de cuatro fases, se muestra por qué es difícil detectar y eliminar errores, por qué es complejo el proceso de realizar pruebas, y por qué es necesario prestarle mayor atención.

[1] Publicado en la Revista de Ingeniería 35, 34-40. 2011.

INTRODUCCIÓN

Toda empresa que desarrolla software prueba sus productos, pero, aun así, antes de entregarlos siempre contienen anomalías residuales de diversa gravedad. A veces es difícil imaginar cómo es que los probadores no detectan algunos errores evidentes, y en muchas empresas los probadores están mal preparados para ejecutar la difícil tarea de ensayar productos software cada vez más complejos. Los resultados en muchas encuestas informales, hechas a los asistentes a seminarios, sugieren que algunos de los que realizan pruebas, como profesión o como un complemento al desarrollo, tienen un adecuado entrenamiento en pruebas o tienen acceso a buenos libros de pruebas de software.

James Whittaker [1] ofrece algunas luces acerca de por qué el proceso de probar el software es tan retador, e identifica varios enfoques concretos que todos los probadores deberían ser capaces de aplicar fácilmente: el probador eficiente tiene a su disposición un amplio conjunto de técnicas de prueba, entiende cómo se utilizará el producto en su entorno operativo, tiene buen olfato para encontrar errores sutiles y tiene a la mano una bolsa de trucos que sabe utilizar. Los métodos que se describe en este trabajo pueden ayudar a los probadores para dar una respuesta sensata a la cuestión de lo que realmente quieren expresar cuando dicen: *estamos ejecutando pruebas a un sistema software*.

Los desarrolladores conocen la frustración cuando reciben reportes de errores de parte de los usuarios. Cuando esto sucede inevitablemente se preguntan: ¿cómo escaparon esos errores a las pruebas? Sin duda que invirtieron incontables horas en el examen cuidadoso de cientos o miles de variables y sentencias de código, así que ¿cómo puede un error eludir esta vigilancia? La respuesta requiere, en primer lugar, una mirada más atenta a las pruebas de software en el contexto de desarrollo y, en segundo lugar, comprender el papel que juegan los probadores y los desarrolladores en dicho contexto, dos funciones parecidas, pero muy diferentes.

Suponiendo que las anomalías que reportan los usuarios realmente son errores, la respuesta a la anterior pregunta podría ser cualquiera de las siguientes:

- *El usuario ejecuta un código no probado*. Por falta de tiempo, no es raro que los desarrolladores liberen código sin probar, en el que los usuarios pueden encontrar anomalías.
- *El orden en que se ejecuta las declaraciones en el ambiente de uso difiere del que se utilizó durante la prueba*. Este orden puede determinar si el software funciona bien o no.
- *El usuario aplica una combinación de valores de entrada no probados*. Las posibles combinaciones de valores de entrada, que miles de usuarios pueden hacer a través de una interfaz de software, simplemente son numerosas para que los probadores las apliquen todas y, como deben tomar decisiones difíciles acerca de qué valores de entrada probar, a veces toman las equivocadas.
- *El entorno operativo de usuario nunca se probó*. Es posible que los probadores tengan conocimiento de dicho entorno, pero que no cuenten con el tiempo suficiente para probarlo. Tal vez no replicaron la combinación de hardware, periféricos, sistemas operativos y aplicaciones del entorno del usuario en el laboratorio de pruebas. Por ejemplo, aunque es poco probable que las empresas que escriben software creen redes de miles de nodos en su laboratorio de pruebas, los usuarios sí lo hacen y, de hecho, lo hacen en sus entornos reales.

Desde una visión general del problema y del proceso de las pruebas del software, en este capítulo se analiza y describe los problemas que enfrentan los probadores, e identifica las cuestiones

técnicas que cualquier solución de prueba debe abordar. Además, estudia las clases de soluciones que se utiliza en la práctica.

1. EL PROCESOS DE PRUEBA

Al planificar y ejecutar las pruebas los probadores de software deben considerar: el software y su función de cálculo, las entradas y cómo se pueden combinar, y el entorno en el que el software eventualmente funcionará. Este difícil proceso requiere tiempo, sofisticación técnica y una adecuada planificación. Los probadores no solo deben tener buenas habilidades de desarrollo (a menudo las pruebas requieren una amplia cantidad de código), sino también conocimientos en lenguajes formales, teoría de grafos, lógica computacional y algoritmia. De hecho, los probadores creativos aplican muchas disciplinas relacionadas con la informática al problema de las pruebas, a menudo con resultados impresionantes.

Incluso el software más simple presenta obstáculos por lo que, para tener una visión más clara acerca de las dificultades inherentes a las pruebas, es necesario acercarse a ellas a través de la aplicación de cuatro fases, que les ofrezcan a los probadores una estructura en la que puedan agrupar los problemas relacionados que deben resolver antes de pasar a la siguiente fase.

1.1 FASE 1: Modelar el entorno del software

La tarea del probador es simular la interacción entre el software y su entorno, para lo que debe identificar y simular las interfaces que utiliza el sistema, y enumerar las entradas que pueden circular por cada una de ellas. Éste podría ser el asunto más importante que enfrentan y, teniendo en cuenta los diversos formatos de archivo, los protocolos de comunicación y las terceras partes disponibles (interfaces de programación de las aplicaciones) puede ser muy complicado. Las interfaces más comunes son:

- *Humanas*: incluyen todos los métodos comunes con los que las personas se comunican con el software. La más destacada es la interfaz gráfica de usuario GUI, pero todavía se utiliza antiguos diseños como la interfaz de línea de comandos y la basada en menús. Los posibles mecanismos de entrada que se deben considerar son los *clics*, pulsaciones de teclado y entradas desde otros dispositivos. Los probadores deciden entonces cómo organizar estos datos para comprender cómo ensamblarlos en una prueba efectiva.

- *De software*: llamadas Application Programming Interfaces APIs, indican cómo utiliza el software al sistema operativo, la base de datos o las librerías, en tiempo de ejecución. Los servicios que estas aplicaciones ofrecen se modelan como entradas de prueba, pero el desafío para los probadores es comprobar, no solo las probables, sino también las inesperadas. Por ejemplo, todos los desarrolladores esperan que el sistema operativo guarde los archivos por ellos, pero olvidan que les puede informar que el medio de almacenamiento está lleno, por lo que, incluso, los mensajes de error deben probarse.

- *Del sistema de archivos*: existen siempre que el software lea o escriba datos en archivos externos. Los desarrolladores deben escribir líneas de comprobación de errores para determinar si el archivo contiene datos y formato adecuados. Por lo tanto, deben generar archivos con contenido, que a la vez sea legal e ilegal, y que contengan texto y formato variados.

- *Las interfaces de comunicación*, permiten el acceso directo a dispositivos físicos, como los controladores de dispositivos y otros sistemas embebidos, y requieren un protocolo de

comunicación específico. Los probadores deben ser capaces de generar protocolos válidos e inválidos para examinarlas; además de poder ensamblar muchas y diferentes combinaciones de comandos y datos para aplicarlos a la interfaz bajo prueba, en el formato del paquete apropiado.

Luego los probadores deben comprender las interacciones de usuario que están fuera del control del software bajo prueba, ya que si el software no está preparado las consecuencias pueden ser graves. Ejemplos de situaciones que los probadores deben abordar son:

- Usando el sistema operativo un usuario elimina un archivo que otro usuario tenía abierto, ¿qué pasará la próxima vez que el software intente accesar ese archivo?
- Un dispositivo se reinicia en medio de un proceso de comunicación, ¿podrá el software darse cuenta de esto y reaccionar adecuadamente, o simplemente lo dejará pasar?
- Dos sistemas compiten por duplicar servicios desde la API, ¿podrá la API atender correctamente ambos servicios?

Cada entorno único de aplicación puede resultar en un número significativo de interacciones de usuario que se debe probar.

1.1.1 Para tener en cuenta

Cuando una interfaz presenta problemas de tamaño o de complejidad infinitos, los probadores se enfrentan a dos dificultades: 1) seleccionar cuidadosamente los valores para cualquier variable de entrada, y 2) decidir cuál será la secuencia de las entradas. En la selección de valores deben determinar el de las variables individuales, y asignar las combinaciones adecuadas cuando el programa acepta múltiples variables como entrada.

Frecuentemente utilizan la técnica Boundary Value Partitioning [2] para seleccionar valores individuales para las variables, en o alrededor de sus fronteras. Por ejemplo, probar los valores máximos, mínimos y cero para un entero con signo es una prueba común, lo mismo que los valores que rodean cada una de estas particiones, por ejemplo, 1 y -1 que rodean la frontera cero. Los valores entre las fronteras se tratan como el mismo número: utilizar 16 o 16.000 no hace ninguna diferencia para el software bajo prueba.

Una cuestión más complicada es elegir los valores para múltiples variables procesadas simultáneamente y que potencialmente podrían afectar a otras, para lo que deben considerar el producto completo que resulta de la combinación de valores. Por ejemplo, para dos números enteros: considerar ambos positivos, ambos negativos, uno positivo y uno cero, y así sucesivamente [3]. Al decidir cómo será la secuencia de entrada, los probadores tienen un problema de generación de secuencia, por lo que deben tratar cada entrada física y cada evento abstracto como símbolos en el alfabeto de un lenguaje formal, definir un modelo de ese lenguaje que les permita visualizar el posible conjunto de pruebas, e indagar cómo se ajusta cada una a la prueba general.

El modelo más común es un grafo o diagrama de estados, aunque existe muchas variaciones: otros modelos populares incluyen expresiones regulares y gramaticales, herramientas de la teoría de lenguajes; menos utilizados son los modelos estocásticos de procesos y los algoritmos genéticos, pero, en general, el modelo es una representación que describe cómo se combina las entradas y los símbolos de los eventos para formar palabras y oraciones sintácticamente válidas.

Esas oraciones son secuencias de entrada que se pueden aplicar al software bajo prueba. Un ejemplo de esto es considerar la entrada *Filemenu.Open*, que involucra una caja de dialogo para seleccionar archivos; *Filename*, que representa la selección (tal vez con *clics*) de un archivo existente, y *ClickOpen* y *ClickCancel*, que representan el botón accionado. La secuencia *Filemenu.Open Filename ClickOpen* es correcta, como muchas otras, pero la secuencia *ClickCancel Filemenu.Open* es incorrecta, ya que el botón de cancelación no se puede presionar hasta que la caja de diálogo se haya invocado. Un modelo en lenguaje formal puede hacer una distinción entre las secuencias a aplicar.

Podemos representar el uso correcto de la caja de diálogo para seleccionar archivos, por ejemplo, en un editor de texto, con la expresión: *Filemenu.Open filename* ClickOpen|ClickCancel*, en la que el asterisco representa el operador de clausura de Kleene [4], y demuestra que la acción *filename* puede ocurrir cero o más veces. Esta expresión indica que la primera entrada recibida es *Filemenu.Open*, seguida de cero o más selecciones de un *filename* (una combinación de *clics* y entradas de teclado) y que, a continuación, se presiona el botón *Open* o *Cancel*. Este sencillo modelo representa todas las combinaciones de entrada que pueden suceder y si tienen sentido o no. Para completarlo tendríamos que representar secuencias para las interfaces de usuario y del sistema operativo. Además, necesitaríamos una descripción de los archivos legales y corruptos para investigar a fondo la interacción del sistema de archivos, tarea que requiere usar ampliamente la lógica, la descomposición y la abstracción.

1.2 FASE 2: Seleccionar escenarios de prueba

Muchos modelos de dominio y particiones de variables representan un número infinito de escenarios de prueba, cada uno de los cuales cuesta tiempo y dinero. Solo un sub-conjunto de ellos se puede aplicar en cualquier programa de desarrollo de software realista, así que ¿cómo hace un probador inteligente para seleccionar ese sub-conjunto? ¿17 es mejor valor de entrada que 34? ¿cuántas veces se debe seleccionar un *filename* antes de pulsar el botón *Open*? Estas cuestiones, que tienen muchas respuestas, todavía se investigan activamente. Sin embargo, los probadores prefieren una respuesta que se refiera a la cobertura de código fuente, o a su dominio de entrada, y se orientan por la cobertura de las declaraciones de código (ejecutar cada línea de código fuente por lo menos una vez), o la cobertura de entradas (aplicar cada evento generado externamente). Estos son los criterios mínimos que utilizan para juzgar la completitud de su trabajo, por lo tanto, el conjunto de casos de prueba que muchos eligen es el que cumpla con sus metas de cobertura.

Pero si el código y la cobertura de entrada son suficientes los productos entregados deberían tener muy pocos errores. En cuanto al código, no son las declaraciones individuales las que interesan a los probadores, sino los caminos de ejecución: secuencias de declaraciones de código que representan un camino de ejecución del software, pero, desafortunadamente, existe un número infinito de caminos. En cuanto al dominio de entrada, no les interesan las individuales, sino las secuencias de entrada que, en conjunto, representen escenarios a los que el software debe responder, pero también existe un número infinito de ellas.

Las pruebas se organizan desde dichos conjuntos infinitos hasta lograr, lo mejor posible, los criterios adecuados de datos de prueba; que se utilizan adecuada y económicamente para representar cualquiera de esos conjuntos. *Mejor* y *adecuadamente* son cuestiones subjetivas: los probadores típicamente buscan el conjunto que garantizará encontrar la mayoría de los errores. Muchos usuarios y profesionales de aseguramiento de la calidad del software están interesados en que los probadores evalúen los escenarios de uso típicos (cosas que ocurren con mayor

frecuencia en el uso del producto). Probar esos escenarios puede asegurar que el software funciona de acuerdo con lo especificado, y que se ha detectado los errores más frecuentes.

Para citar un caso, consideremos nuevamente el ejemplo del editor de texto: para probar el uso típico nos centraremos en la edición y el formato, puesto que es lo que la mayoría de usuarios reales hace; no obstante, para encontrar errores, un lugar con mayor probabilidad son las características más difíciles de código, como el dibujo de figuras y la edición de tablas.

1.2.1 Criterios de prueba de los caminos de ejecución

Los criterios adecuados para datos de prueba se concentran en la cobertura de caminos de ejecución o en la cobertura de secuencias de entrada, pero rara vez en ambos. El criterio de selección de caminos de ejecución más común es el de aquellos que cubran las estructuras de control. Por ejemplo: 1) seleccionar un conjunto de casos de prueba que garantice que cada sentencia se ejecute al menos una vez, y 2) seleccionar un conjunto de casos de prueba que garantice que cada estructura de control *If, Case, While, …* se evalúe en cada uno de sus posibles caminos de ejecución. Sin embargo, el flujo de control es solo un aspecto del código fuente. Entonces, ¿qué software mueve datos de un lugar a otro? La familia de flujo de datos del criterio, adecuada para datos de prueba, describe la cobertura de estos datos [5] como el seleccionar un conjunto de casos de prueba que garantice que cada estructura de datos se inicialice, y que posteriormente se utilice.

Por otro lado, la siembra de errores [2] es interesante, aunque tiene más atención de los investigadores que de los probadores. En este método los errores se añaden intencionadamente en el código fuente, y para encontrarlos se diseñan escenarios de prueba. Lo ideal sería que al encontrarlos también se encontraran errores reales. Por lo tanto, es posible un criterio como: seleccionar un conjunto de casos de prueba que exponga cada uno de los errores sembrados.

1.2.2 Criterios de prueba del dominio de entrada

El criterio para el rango de cobertura del dominio abarca desde la cobertura de una interfaz sencilla hasta la medición estadística más compleja:

- Elegir un conjunto de casos de prueba que contenga cada entrada física.
- Seleccionar un conjunto de casos de prueba que garantice que se recorra cada interfaz de control.

El criterio de discriminación requiere una selección aleatoria de secuencias de entrada hasta que, estadísticamente, representen todo el dominio infinito de las entradas:

- Seleccionar un conjunto de casos de prueba que tenga las mismas propiedades estadísticas que el dominio de entrada completo.
- Escoger un conjunto de rutas que puedan ser ejecutadas por un usuario típico.

Es decir, los investigadores de pruebas estudian algoritmos para seleccionar conjuntos de prueba mínimos que cumplan los criterios para caminos de ejecución y dominios de entrada. La mayoría de ellos está de acuerdo en que es prudente utilizar varios criterios cuando se toman decisiones importantes para cada versión del producto. Los experimentos para comparar criterios adecuados para datos de prueba son necesarios, así como los nuevos criterios. No obstante, por

ahora, los probadores deben estar conscientes de qué criterios integrar en su metodología y comprender las limitaciones inherentes de esos criterios cuando reportan resultados.

1.3 FASE 3: Ejecutar y evaluar los escenarios

Una vez identificado el conjunto de casos de prueba adecuado, los probadores lo convierten a formatos ejecutables, a menudo código, de modo que los escenarios de prueba resultantes simulen la acción de un usuario típico. Debido a que los escenarios de prueba se ejecutan manualmente constituye un trabajo intensivo y, por tanto, propenso a errores, por lo que los probadores deben tratar de automatizarlos tanto como sea posible. En muchos entornos es posible aplicar automáticamente las entradas a través del código que simula la acción de los usuarios, y existe herramientas que ayudan a este objetivo. Pero la automatización completa requiere la simulación de cada fuente de entrada, y del destino de la salida de todo el entorno operacional. A menudo, los probadores incluyen código para recoger datos en el entorno simulado, como ganchos o seguros de prueba, con los que recogen información acerca de las variables internas, las propiedades del objeto, y otros. Esto ayuda a identificar anomalías y a aislar errores. Estos ganchos se eliminan cuando el software se entrega.

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar, pero difícil de ejecutar porque es menos automatizada. La evaluación implica la comparación de las salidas reales del software, resultado de la ejecución de los escenarios de prueba, con las salidas esperadas, tal y como están documentadas en la especificación, que se supone correcta, ya que las desviaciones son errores. En la práctica esta comparación es difícil de lograr. Teóricamente, la comparación (para determinar la equivalencia) de dos funciones arbitrarias computables es irresoluble. Volviendo al ejemplo del editor de texto: si la salida se supone que es *resaltar una palabra mal escrita*, ¿cómo se puede determinar que se ha detectado cada instancia de errores ortográficos? Tal dificultad es la razón por la que la comparación de la salida real versus la esperada se realiza generalmente por un oráculo humano: un probador que monitorea visualmente la pantalla de salida y analiza cuidadosamente los datos que aparecen.

1.3.1 Enfoques para evaluar las pruebas

Al tratar con el problema de la evaluación de la prueba los investigadores aplican dos enfoques: la formalización y el código de prueba embebido.

- La formalización consiste en *formalizar* el proceso de escritura de las especificaciones y la forma cómo, desde ellas, se derivan el diseño y el código [6]. Tanto el desarrollo orientado por objetos, como el estructurado, tienen mecanismos para expresar formalmente las especificaciones, de forma que se simplifique la tarea de comparar el comportamiento real y el esperado. La industria generalmente les ha rehuido a los métodos formales; no obstante, una buena especificación, aunque informal, sigue siendo de gran ayuda. Sin una especificación posiblemente los probadores pueden encontrar solo los errores más obvios. Por otra parte, la ausencia de una especificación redundaría en una pérdida significativa de tiempo cuando se reporten, como errores, anomalías no especificadas.

- Esencialmente, existe dos tipos de código de prueba embebido: 1) el más simple es el código de prueba que expone algunos objetos de datos internos, o estados, de tal forma que un oráculo externo pueda juzgar su correctitud más fácilmente. Al implementarse, dicha funcionalidad es invisible para los usuarios. Los probadores pueden tener acceso a resultados del código de prueba a través de, por ejemplo, una prueba al API o a un depurador. 2) Un tipo

más complejo de código embebido tiene características de programa de auto-prueba [7]: a veces se trata de soluciones de codificación múltiple para el problema, para chequear o escribir rutinas inversas que deshacen cada operación. Si se realiza una operación y luego se deshace, el estado resultante debe ser equivalente al estado pre-operacional. En esta situación el oráculo no es perfecto y podría haber errores enmascarados.

1.3.2 Pruebas de regresión

Después de que los probadores presentan con éxito los errores encontrados, generalmente los desarrolladores crean una nueva versión del software, en la que, supuestamente esos errores se han eliminado. La prueba progresa a través de versiones posteriores del software hasta una que se selecciona para entregar. La pregunta es: ¿cuántas re-pruebas, llamadas pruebas de regresión, se necesitan en la versión *n*, cuando se re-utilizan las pruebas ejecutadas sobre la versión *n-1*?

Cualquier selección puede: 1) corregir solo el problema que fue reportado, 2) fallar al corregir el problema reportado, 3) corregir el problema reportado, pero interrumpir un proceso que antes trabajaba, o 4) fallar al corregir el problema reportado e interrumpir un proceso funcional. Teniendo en cuenta estas posibilidades sería prudente volver a ejecutar todas las pruebas de la versión *n-1* en la versión *n*, antes de probar nuevamente, aunque esta práctica generalmente tiene un costo elevado [8]. Por otra parte, las nuevas versiones del software a menudo vienen con características y funcionalidades nuevas, además de los errores corregidos, así que las pruebas de regresión les quitarían tiempo a las pruebas del código nuevo. Para ahorrar recursos los probadores deben trabajar en estrecha colaboración con los desarrolladores, para establecer prioridades y reducir al mínimo las pruebas de regresión.

Otro inconveniente de estas pruebas es que pueden, temporalmente, modificar el criterio adecuado seleccionado en la fase anterior para datos de prueba. Cuando se realiza pruebas de regresión, los probadores solo pretenden demostrar la ausencia de errores y forzar su aplicación a que exhiba un comportamiento específico. El resultado es que el criterio adecuado para datos de prueba, que hasta ahora guía la selección de los casos de prueba, es ignorado. Por lo que, en su lugar, los probadores deben asegurarse de que el código se haya corregido adecuadamente.

1.3.3 Asuntos relacionados

Idealmente, los desarrolladores deberían escribir código teniendo en su mente las pruebas: si el código va a ser difícil de verificar y validar, entonces se debería reescribir de tal forma que pueda verificarse y validarse adecuadamente. Del mismo modo, una metodología de prueba debería juzgarse de acuerdo con su contribución a la automatización y al oráculo de la solución de los problemas. Muchas metodologías propuestas ofrecen poca orientación en cualquier de estas áreas. Otra preocupación para los probadores, mientras ejecutan pruebas de verificación o validación, es cómo coordinar con los desarrolladores las actividades de depuración. Dado que los errores los identifican los probadores, pero los diagnostican los desarrolladores, puede suceder: 1) que su reproducción fracase, o 2) que el escenario de prueba se ejecute nuevamente.

Que fracase la reproducción no es tan simple como parece, por lo que la respuesta obvia sería, por supuesto, volver a ejecutar la prueba fracasada y observar nuevamente el comportamiento de los resultados, aunque volver a efectuar una prueba no garantiza que se reproduzcan las mismas condiciones originales. Re-ejecutar un escenario requiere conocer con exactitud el estado del sistema operativo y cualquier software que lo acompañe (tal es el caso de las aplicaciones cliente-servidor que requieren la reproducción de las condiciones del entorno, tanto en el cliente

como en el servidor). Además, hay que conocer el estado de automatización de la prueba, los dispositivos periféricos y cualquiera otra aplicación de segundo plano, que se ejecute localmente o través de la red y que podría afectar la aplicación bajo prueba. No es de extrañar que una de las frases que comúnmente se escucha en los laboratorios de prueba es: *el programa se comporta de forma diferente antes de…*

1.4 FASE 4: Medir el progreso de las pruebas

Supongamos que cualquier día el jefe de un probador viene y le pregunta: ¿cuál es el estado de sus pruebas? Los probadores escuchan a menudo esta pregunta, pero no están bien preparados para responderla. La razón es que, en la práctica, medir las pruebas consiste en *contar cosas*: el número de entradas aplicadas, el porcentaje de código cubierto, el número de veces que se ha invocado la aplicación, el número de veces que se ha terminado la aplicación con éxito, el número de errores encontrados, y así sucesivamente.

La interpretación de estas *cuentas* es difícil: ¿encontrar un montón de errores es buena o mala noticia? La respuesta podría ser: un alto número de errores significa que la prueba se ejecutó completamente y que persisten muy pocos errores; o simplemente que el software tiene un montón de errores y que, a pesar de que muchos fueron encontrados, otros permanecen ocultos.

Los valores de estos conteos pueden dar muy pocas luces acerca de los avances de las pruebas, y muchos probadores alteran estos datos para dar respuesta a las preguntas, con lo que determinan la completitud estructural y funcional de lo que han hecho. Por ejemplo, para comprobar la completitud estructural los probadores pueden hacerse preguntas como:

- ¿He probado para errores de programación común? [9]
- ¿He ejercitado todo el código fuente? [2]
- ¿He forzado a todos los datos internos a ser inicializados y utilizados? [5]
- ¿He encontrado todos los errores sembrados? [2]

Y para probar la completitud funcional:

- ¿He tenido en cuenta todas las formas en las que el software puede fallar y he seleccionado casos de prueba que las muestren y casos que no lo hagan? [9]
- ¿He aplicado todas las posibles entradas? [2]
- ¿Tengo completamente explorado el dominio de los estados del software? [1]
- ¿He ejecutado todos los escenarios que espero que un usuario ejecute? [10]

Estas preguntas, fundamentalmente el criterio adecuado para datos de prueba, son útiles para los probadores, pero determinar cuándo detener las pruebas o cuándo está listo un producto para su liberación es más complejo. Se necesitan medidas cuantitativas de la cantidad de errores que queden en el software, y de la probabilidad de que cualquiera de ellos sea descubierto por el usuario. Si los probadores pudieran lograr esta medida sabrían cuándo parar las pruebas, y sería posible que se acercaran cuantitativamente al problema de forma estructural y funcional.

1.4.1 Capacidad de prueba

Desde un punto de vista estructural, Jeffrey Voas [11] propuso la capacidad de prueba como una manera para determinar la complejidad de la aplicación de una prueba: la idea de que el número

de líneas de código determine la dificultad de la prueba es obsoleta, la cuestión es mucho más complicada y es donde entra en juego la capacidad de prueba. Si un producto software tiene alta capacidad de prueba: 1) será más fácil de probar y, por consiguiente, más fácil de encontrar sus errores, y 2) será posible monitorear las pruebas, por lo que los errores y disminuyen las probabilidades de que se queden otros sin descubrir. Una baja capacidad de prueba requerirá muchas más pruebas para llegar a estas mismas conclusiones, y es de esperar que sea más difícil encontrar errores. La capacidad de prueba es un concepto convincente, pero todavía no se ha publicado suficientes datos acerca de su capacidad predictiva.

1.4.2 Modelos de fiabilidad

¿Cuánto durará el software en ejecución antes de que falle? ¿Cuánto costará el mantenimiento del software? Sin duda es mejor encontrar respuestas a estas preguntas mientras todavía se tenga el software en el laboratorio de pruebas. Desde un punto de vista funcional los modelos de fiabilidad [10] (modelos matemáticos de escenarios de prueba y datos de errores) están bien establecidos. Con base en cómo se comportó durante las pruebas, estos modelos pretenden predecir cómo se comportará el software en su entorno funcional. Para lograrlo la mayoría de ellos requieren la especificación de un perfil operativo: una descripción de cómo se espera que los usuarios apliquen las entradas.

Para calcular la probabilidad de una falla estos modelos hacen algunas suposiciones acerca de la distribución de probabilidades subyacentes que regulan las ocurrencias de las anomalías. Tanto los investigadores como los probadores expresan escepticismo acerca de que se pueda ensamblar adecuadamente estos perfiles. Por otra parte, las hipótesis hechas por los modelos de fiabilidad aún no se verifican teórica o experimentalmente, salvo en dominios n específicos. Sin embargo, estudios de caso exitosos demuestran que estos modelos pueden ser creíbles.

2. CONCLUSIONES

Las compañías de software se enfrentan con serios desafíos cada vez más grandes en la prueba de sus productos debido a que cada día se incrementa la complejidad del software.

Lo primero y más importante que hay que hacer es reconocer la naturaleza compleja de las pruebas y tomarlas en serio: contratar a las personas más inteligentes que se pueda encontrar, ayudarlas a conseguir y/o brindarles las herramientas y el entrenamiento que requieran para aprender su oficio, y escucharlos cuando hablan acerca de la calidad del software. Ignorarlos podría ser el error más costoso para la empresa y el producto.

Los investigadores en pruebas se enfrentan igualmente a estos desafíos. Las compañías están ansiosas por financiar buenas ideas de investigación, pero la demanda fuerte es por más práctica experimental y menos trabajo académico. El tiempo para concatenar la investigación académica con los productos industriales es ahora.

Las cuatro fases que estructuran la metodología propuesta en este capítulo no se deben considerar como definitivas y únicas, porque este es un campo en constante desarrollo e investigación en el que están involucrados muchos investigadores y empresas.

REFERENCIAS

[1] Whittaker J. y Thomason M. (1994). A Markov chain model for statistical software testing. IEEE Transactions on Software Engineering 20(10), 812-824.

[2] Myers G. (1976). The Art of Software Testing. John Wiley.
[3] Ostrand T. y Balcer M. (1988). The Category-Partition Technique for Specifying and Generating Functional Tests. Communications of the ACM 31(6), 676-686.
[4] Kozen, D. (2002). On Hoare Logic, Kleene Algebra and Types. Studies in Epistemology, Logic, Methodology and Philosophy of Science 315, 119-133.
[5] Rapps S. y Weyuker E. (1985). Selecting Software Test Data Using Dataflow Information. IEEE Transactions on Software Engineering 11(4), 367-375.
[6] Peters D. y Parnas D. (1998). Using Test Oracles Generated from Program Documentation. IEEE Transactions on Software Engineering 24(3), 161-173.
[7] Knuth D. (1984). Literate Programming. The Computer Journal 27(2), 97-111.
[8] Rothermel G. y Harrold M. (1993). A Safe, Efficient Algorithm for Regression Test Selection. En Conference on Software Maintenance. Montreal, Canada.
[9] Goodenough J. y Gerhart S. (1975). Toward a Theory of Test Data Selection. IEEE Transactions on Software Engineering 2(2), 156-173.
[10] Musa J. (1996). Software Reliability Engineered Testing. Computer 29(11), 61-68.
[11] Voas J. (1992). PIE: A Dynamic Failure-Based Technique. IEEE Transactions on Software Engineering 18(8), 717-727.

CAPÍTULO XXIV
Una evaluación a las herramientas libres para pruebas de software[1]

Edgar Serna M.
Alexei Serna A.
Instituto Antioqueño de Investigación

Cuando se inicia un proyecto software una de las primeras actividades consiste en seleccionar una herramienta de prueba. Debido a que algunas de ellas son multifuncionales, producidas para propósitos limitados, de código abierto o tienen un costo elevado, se cuenta con un espacio multidimensional de métricas para resolver el problema de la selección. Pero, antes de elegir una herramienta de prueba, es prudente pensar primero en los deseos y necesidades del proyecto, para luego comparar algunas candidatas y, por último, probar una o dos de ellas. Al parecer la ausencia de una tarifa de licencia inicial es la justificación necesaria que muchos necesitan para seleccionar una u otra, sin embargo, existe otros factores que influyen en el costo total de una herramienta de prueba. En este trabajo se analiza algunos mitos acerca de las herramientas de prueba de código abierto; se describe sus pros y sus contras; se refiere su evolución hasta convertirse en parte integral de las estrategias de desarrollo; se analiza el efecto que tendrán su evolución en el mercado de las herramientas de prueba; se evalúa algunas de las más populares de estas y se propone algunos trabajos futuros.

[1] Publicado en la Revista Virtual Universidad Católica del Norte 37, 44-61. 2012.

INTRODUCCIÓN

Cuando se realiza un proyecto software una de las actividades es seleccionar una herramienta para las pruebas, por lo tanto, es necesario conocer su contexto. Algunas son multifuncionales, otras se construyen para propósitos limitados, unas son de código abierto y otras tienen un costo bastante elevado para la mayoría de proyectos. Todo esto crea un espacio multidimensional de métricas que se debe tener en cuenta para resolver el problema de la selección. Existe grandes empresas que tienen estándares para definir y seleccionar alguna, pero el interés de este trabajo es diferente.

Antes de seleccionar una herramienta para pruebas es prudente pensar primero en los requisitos y las necesidades del proyecto, luego comparar algunas candidatas y posteriormente probarlas hasta llegar a la selección más adecuada. De alguna manera a menudo las herramientas de código abierto se aceptan sin ningún tipo de proceso de selección previo; al parecer, la ausencia de una tarifa de licencia es la justificación necesaria. Sin embargo, existe otros factores que influyen en el costo total de este tipo de herramientas y que se deben tener en cuenta desde el principio, de lo contrario es posible que se termine pagando demasiado para lograr muy poco.

1. EVOLUCIÓN Y FUTURO DEL CÓDIGO ABIERTO

En esta sección se describe cómo ha evolucionado el software de código abierto desde el dominio de la tecnología de los *geeks*, hasta ser parte integral de las estrategias en software de muchas organizaciones. Debido a que el software de código abierto está migrando de la infraestructura de la empresa al mercado de aplicaciones [1], también se analiza el impacto que esto tendrá en el desarrollo tecnológico de los principales proveedores de herramientas para pruebas.

1.1 Evolución del código abierto

A causa de las recesiones económicas las organizaciones de todo el mundo estrechan sus presupuestos, y la mayor parte de la industria del software y la de servicios sienten sus efectos. Sin embargo, las compañías de código abierto han superado este impacto al mostrar un proceso evolutivo. En medio de la recesión de comienzos del siglo XXI, un artículo en The Economist [2] denominó *zeitgeist* a ese crecimiento. En general ha habido un incremento real de historias de éxito del código abierto en los medios de comunicación, mientras que los resultados de las investigaciones acerca de su adopción, realizadas por analistas de la industria como IDC y Gartner, muestran un fuerte crecimiento de este sector.

En 2008 Gartner encuestó a 274 empresas en el mundo y encontró que el 85% ya había adoptado el código abierto, y que la mayoría restante esperaba hacerlo dentro del siguiente año [3]. Esa predicción se convirtió en realidad y el reporte Forrester de 2009 [4] interrogó a 2.000 tomadores de decisiones de software; la conclusión fue que el código abierto había llegado a la cima de la agenda ejecutiva, como señala el autor: *la orden viene desde los más alto* y muchos jefes están demandando procesos *más rápidos, más baratos y mejores*.

Esa fue una importante etapa de cambio para el código abierto, que había entrado históricamente de abajo-arriba en el radar ejecutivo de las organizaciones, instalado por quipos de TI y desarrolladores, y gradualmente había ganado terreno. En el reporte de Accenture [5] se hizo eco de esto al expresar que las empresas de servicios financieros, mientras estaban limitadas económicamente, empezaron a adoptar un punto de vista diferente acerca de las tecnologías de código abierto. Dicho reporte expresa además que los principales bancos no solo utilizan el código

abierto, sino que contribuyen a proyectos reconociendo un futuro cambio radical y una mayor madurez para su adopción, que va más allá del simple ahorro de costos.

1.2 Impacto del código abierto en el desarrollo tecnológico

Tradicionalmente, gran parte del empuje obtenido por el código abierto ha estado a nivel de infraestructura de TI, sin embargo, otro gran cambio en su adopción se ha dado en la migración desde esa infraestructura a las aplicaciones. Algunos mercados de aplicaciones han sido penetrados significativamente por el código abierto, como SugarCRM en el mercado de CRM y Alfresco en el mercado de la gestión de documentos.

Pero el entorno de las herramientas para pruebas es un caso interesante en el que el código abierto ha tenido un significativo progreso. La organización Open Source Testing inició labores en 2003 con cerca de 50 herramientas en su lista, y en casi diez años ese número se incrementó a más de 450. El incremento en esta actividad es mucho más que lo ocurrido en el mercado CRM, por ejemplo, antes de que Sugar se convirtiera en la aplicación dominante. Sin embargo, el mercado de estas herramientas está más fragmentado en áreas variadas y especializadas, lo que explica el número de productos de código abierto existentes. Aunque en un área específica, como el seguimiento de errores, solo un puñado de ellas compite realmente por la supremacía.

Este mercado es joven y todavía no tiene una inversión comercial importante, pero independientemente de eso se espera que el progreso permanezca constante. La naturaleza influyente del código abierto en los mercados de software ha hecho que los proveedores comerciales evalúen su posición, y la respuesta fue innovar sus productos con mayor rapidez para mantenerse un paso adelante, o innovar su modelo de negocios. Microsoft Sharepoint es ejemplo de un producto comercial que adoptó el segundo enfoque con el objetivo de alcanzar el primero. Su núcleo es como siempre código cerrado, sin embargo, existe más de 3.000 APIs disponibles para que los desarrolladores puedan extender el producto de nuevas e imprevistas formas, creando servicios, complementos y soluciones integradas.

Como se señala en CMS Wire, esto se ha traducido en un ecosistema y en una comunidad vibrante para competir con la de cualquier producto de código abierto, un astuto movimiento por parte de Microsoft. El éxito de Sharepoint será replicado más ampliamente en los próximos años por los vendedores comerciales, frente a la competencia del código abierto. Es probable que las inclinaciones de Mercury (Hewlett Packard) se muevan en esa dirección, lo que más adelante le podría generar logros interesantes e innovadores.

2. MITOS Y REALIDADES DE LAS HERRAMIENTAS LIBRES

Es común escuchar al inicio de los proyectos software que los directores lanzan expresiones como: *Vamos a excluir a las herramientas de código abierto de la lista que posiblemente utilizaremos para las pruebas*, y al preguntarles por qué la respuesta que ofrecen es: *No quiero tener problemas adicionales en el futuro con esas herramientas. Vamos a estar bajo presión en este proyecto, por lo que quiero disminuir los riesgos de software generados por terceros* [6].

Así es como los directores de proyectos viven el proceso del desarrollo de software: se tienen que enfrentar a presupuestos, tiempos, recursos humanos y problemas de motivación, y tratan de comparar su propio proyecto con los resultados obtenidos en alguno otro desarrollado bajo código abierto. Sin embargo, se debe tener en cuenta que cada proyecto es único y utiliza reglas diferentes. Este es el principal malentendido que se presenta [7]. A continuación, se describe los mitos (malentendidos) que normalmente se tiene alrededor de estas herramientas.

2.1 Baja calidad

Estructurar un equipo de trabajo es un arte, como muchos directores lo pueden confirmar. Entonces, cómo pueden personas independientes, que ocasionalmente participan en un proyecto de código abierto, producir productos de alta calidad y trabajar como un equipo. Esta es la base de las dudas acerca de los productos de código abierto, sin embargo, cabe señalar que esas personas están lo suficientemente motivadas para participar en cualquier proyecto, que lo convierten en su propia responsabilidad y lo influencian, que lo ven de forma divertida y que lo asumen como una especie de desafío. Tratan de dar lo mejor de sí, porque lo que hacen es una pieza colaborativa del trabajo. Todos pueden inspeccionar y revisar el producto de los otros miembros del proyecto y cada uno lo comprende con claridad; por lo tanto, no se justifica la preocupación por una *baja* calidad.

2.2 Capacitación

Aquí es necesario ser honestos: ¿quién capacita a los probadores antes de comenzar la construcción de un nuevo plan de pruebas? La realidad es que a menudo se capacitan a través de artículos, *podcasts*, foros y muchas otras ayudas colaborativas de la comunidad. Sí, esa es la verdad. Así que, ¿cuál es la diferencia? Cuando se utiliza una herramienta para pruebas de código abierto, usualmente, se tiene la oportunidad de preguntarle al autor acerca de la misma y de sus características particulares, además, se le puede sugerir nuevas y se puede conseguir un contacto más estrecho con los miembros del proyecto. Este es un beneficio y no se puede des-aprovechar.

2.3 Permanencia

Es factible que esto suceda en todo proyecto que no cumpla con las expectativas de los clientes. La ventaja de las herramientas de código abierto es que existe una estrecha relación con los usuarios finales. Los foros, consejos y otras formas similares de comunicación reflejan las expectativas y su posible desarrollo. Existe diversos productos de código abierto que tienen éxito, como Linux y Open Office, pero aquí no es el espacio para debatir acerca de su supervivencia.

2.4 Actualizaciones

Nuevamente la calidad. Pero, ¿será que los clientes proceden inmediatamente a conseguir una actualización? No, esa es la vida real. Conseguir una actualización para una herramienta de código abierto no es tan fácil ni tan difícil como para cualquier producto comercial. Para corroborarlo solo se necesita dialogar con los desarrolladores. Es casi seguro que utilizan el software de código abierto para propósitos de uso personal y de trabajo, y tal vez sean los evaluadores más activos de estos productos.

Los proyectos tienen un presupuesto limitado en lo que tiene que ver con la capacitación en el uso de herramientas para pruebas, por lo tanto, al disminuir los costos de estas herramientas es posible incrementar el presupuesto en capacitación, y hacerlo más eficiente, más enérgico y más flexible. Además, para lograr productos de calidad se debe buscar el mejoramiento permanente de las pruebas y el proceso de desarrollo, y probar nuevos enfoques y metodologías. Por lo tanto, ese es el momento para ensayar las herramientas de prueba de código abierto. Además, se pueden personalizar para cada proyecto, lo que es importante cuando es tan específico que es difícil encontrar soluciones, o cuando se necesita solo una característica particular y no cientos de características incluidas en otros productos para cubrir aspectos que el proyecto no tiene. En la Tabla 1 se resume los mitos y realidades de las herramientas de prueba de código abierto.

Tabla 1. Mitos y realidades de las herramientas de prueba de código abierto

Aspecto	Mito	Realidad
Calidad	No tienen la misma calidad de su contrapartida comercial.	Los equipos de desarrollo se unen por iniciativa de colaboración simultánea.
Capacitación	No hay forma de capacitar al equipo antes de iniciar el proyecto.	La comunidad tiene múltiples medios colaborativos al servicio de los desarrolladores.
Permanencia	Son factibles de cancelación por falta de apoyo económico	Lo mismo le puede suceder a cualquier proyecto de software comercial. La sostenibilidad se garantiza a través de una comunidad motivada y con retos permanentes.
Actualizaciones	No ofrecen rápidamente paquetes de actualización.	Tampoco lo hacen los productos comerciales, la diferencia es que los usuarios tienen acceso al mismo autor para solicitar asesorías.

3. PROS Y CONTRAS DE LAS HERRAMIENTAS LIBRES

En el proceso para seleccionar una herramienta para pruebas es necesario considerar factores que en el ciclo de vida del proyecto entran a jugar diversos papeles, especialmente durante las fases de implementación y mantenimiento. Cuando estos factores se hayan tenido en cuenta será posible hacer una elección satisfactoria. Desde las fases iniciales del ciclo de vida de un proyecto, hasta el momento en que se implementa, es importante seleccionar una herramienta para pruebas; es un proceso que se hace por tanteo y que es conocido como Prueba De Concepto PDC [8], en el que la herramienta se pone a *prueba*. Cuando en ese proceso se trate por igual a las herramientas comerciales y a las de código abierto, será posible elegir a la que mejor se adapte a cada situación.

Para todo proyecto software es necesario determinar el alcance de las pruebas, es decir, es para un proyecto o para todos los proyectos en la organización, qué tan compleja es la aplicación a probar, existe una sola aplicación bajo prueba o son varias. Además, estas herramientas se necesitan para articularlas en la arquitectura global de las pruebas, y se deben ver como facilitadoras del proceso, no como *respuestas*. Con base en esos requisitos, a continuación, se hace un análisis comparativo entre herramientas comerciales y de código abierto.

3.1 Costos y licencias

Además de los derechos de licencia inicial, los otros costos del código abierto son potencialmente más bajos. Por ejemplo, generalmente tiene menores requisitos de hardware que las alternativas comerciales, como OpenSTA, utilizado para pruebas simples de páginas ASP y que soporta hasta 3.000 usuarios virtuales; mientras que LoadRunner, en la misma configuración, puede operar un máximo de 1.000 usuarios virtuales [9].

La etiqueta de código abierto no implica que el software sea gratuito, porque los desarrolladores pueden, y de hecho lo hacen, cobrar por el mismo. *Código abierto* se refiere a la disponibilidad del código fuente, lo que permite cambiar el programa y adaptarlo a cada situación específica. Este software se distribuye bajo diferentes licencias, por ejemplo, Berkeley Software Distribution BSD le permite a cualquier persona modificar el código fuente sin ningún tipo de obligaciones, mientras que General Public License GPL indica que los cambios en el código fuente se deben entregar a la comunidad.

Linksys es un ejemplo de lo que puede suceder si se utiliza código modificado y no se respeta la GPL [10]. En 2003 la empresa lanzó el *router* inalámbrico WRT54G y algunas personas de Linux

Kernel Mailing List lo revisaron, y encontraron que su *firmware* se basaba en componentes Linux. Debido a que Linux se distribuye bajo GPL, los términos de esta licencia obligaron a que Linksys dejara disponible ese código. No es claro si Linksys estaba consciente de la herencia Linux del WRT54G y sus requisitos fuente asociados al momento en que lanzó el *router*, pero en última instancia, bajo la presión de la comunidad del código abierto, tuvo que liberar el *firmware*. Con el código en la mano los desarrolladores aprenden exactamente cómo hablar con el hardware dentro del *router*, y la forma en que las características adicionales del código pueden apoyarlo.

Si algún desarrollador tiene planes de modificar el código fuente de una herramienta de código abierto, para aplicarla en funciones personalizadas, debe saber bajo qué licencia fue liberado. Si el software comercial requiere un valor de licencia, el software de código abierto puede venir con obligaciones de otro tipo.

3.2 Plataformas y características

Usualmente, el software comercial maduro ofrece más características que sus contrapartes de código abierto, que van desde una instalación más fácil y una mejor secuencia de comandos manuales, hasta ejemplos de *scripts* y reportes de fantasía que, a menudo, soportan una amplia gama de plataformas.

Algunas herramientas de código abierto solo pueden probar aplicaciones basadas en web, mientras que las herramientas comerciales también se pueden utilizar con aplicaciones cliente-servidor local. Cada vez es más común que los sistemas adelantados utilicen protocolos de Internet como HTTP/S, IMAP y POP3, sin embargo, a menudo los menos adelantados están conformados por un *mainframe* viejo, pero en buen estado, que puede trabajar por diez o más años.

Una de las características importante de una herramienta es el soporte que trae asociado y, cuando surgen problemas, el proveedor tendrá que resolverlo para el cliente. Este proceso puede tardar algún tiempo, pero conocerlo mantiene tranquilo al cliente. Con herramientas de código abierto él es el que debe solucionar el problema. La comunidad puede o no ayudarle, pero el peso de la solución está sobre sus hombros. Al elegir una de estas herramientas para automatizar las pruebas se debe estar preparado para soportar el tiempo necesario, antes de conseguir que funcione plenamente. Si se subestima este esfuerzo extra, la herramienta puede llegar a ser más costosa que su contraparte comercial. Por otro lado, si se cuenta con la capacidad para resolver cualquier problema que surja, ¿por qué pagar por características que no son necesarias?

Los usuarios admiten que para implementar y mantener herramientas de código abierto se requiere más habilidad y más ingeniería, en comparación con lo requerido por las comerciales. Pero, después de una inversión en la capacitación inicial para adquirir esas habilidades, los costos a largo plazo son más bajos. Obviamente, eliminando el valor de las licencias se ahorra dinero para rublos como la capacitación, sin embargo, encontrarla para herramientas de código abierto es más difícil que para las comerciales, y solo algunas tienen programas de certificación.

El código abierto se vuelve más importante cada vez que se hace un PDC, porque no siempre está bien documentado y puede ser más difícil de instalar. El nivel de habilidad técnica requerido para instalar una de estas herramientas podría ser mayor, debido a que muchos proyectos no se centran en los asistentes de instalación. Una vez instaladas, estas herramientas se actualizan automáticamente, mientras que con otras el seguimiento a las actualizaciones se debe realizar manualmente.

3.3 Probadores y desarrolladores

Las herramientas comerciales apelan a los probadores con la promesa de que no requieren secuencias de comandos. Dado que esas secuencias son su negocio principal, los desarrolladores no son susceptibles a esta promesa, y tienden a preferir las herramientas de código abierto para automatizar las pruebas, porque cuentan con las habilidades y los conocimientos suficientes para ajustarlas a sus necesidades.

En un entorno de desarrollo de código abierto los probadores y los desarrolladores cooperan más estrechamente que en un entorno en cascada. Los primeros ayudan a los segundos a escribir las pruebas unitarias y éstos les ayudan a automatizar las pruebas funcionales. La prueba se considera como una parte integral del proceso de producción y como responsabilidad de cada miembro del equipo. En este contexto el papel del probador cambia de *portero* a *jugador de campo*.

La automatización de las pruebas y la integración continua de nuevo código son prácticas esenciales en los proyectos de código abierto. Parte del ciclo de desarrollo puede ser la primera prueba, es decir, las pruebas se escriben antes que el código, por lo que los desarrolladores escriben código que pasa las pruebas. Posteriormente, se integra en el producto y las pruebas se añaden a la *suite* de regresión, que se aplica automáticamente al nuevo producto. El código base y la *suite* de pruebas crecen de forma altamente interactiva.

Estos proyectos utilizan herramientas de código abierto para sus propósitos de automatización de las pruebas, lo que se puede explicar por el hecho de que esa actividad no es responsabilidad exclusiva de los probadores, sino que es conjunta entre probadores y desarrolladores, trabajando juntos y en equipos multi-funcionales y transdisciplinares.

3.4 Objetos y protocolos

Existe una diferencia fundamental entre las herramientas para automatización de pruebas y las de pruebas de rendimiento. Las primeras trabajan a nivel de la interfaz gráfica de usuario GUI, mientras que las otras lo hacen a nivel de protocolo. La pregunta para las de automatización es si reconocen correctamente todos los objetos en las ventanas de aplicaciones diferentes, y el porcentaje logrado en las respuestas no es del 100%. Si ese reconocimiento es inferior al 50%, los ingenieros de automatización se verán forzados a ejecutar tantas soluciones que no alcanzarán el objetivo de la misma [11], y establecer y mantener la herramienta tomará más tiempo que el requerido para hacerlo manualmente.

Para una herramienta de pruebas de rendimiento la pregunta es si reconoce correctamente los protocolos de comunicación cliente-servidor. Esta pregunta solo se puede responder haciendo un PDC en su propio entorno, y con la aplicación misma. Incluso si la documentación de una herramienta, comercial o de código abierto, afirma que soporta el protocolo, se puede encontrar que no es así, por ejemplo, debido a las diferencias entre versiones.

3.5 Herramientas e infraestructura

Un PDC también es útil para comparar los resultados de las pruebas de diferentes herramientas, porque también son software y contienen errores que pueden aparecer en cualquier momento. En un experimento con WebLOAD, OpenSTA y LoadRunner, instaladas en el mismo equipo y ejecutando el mismo *script* en un laboratorio aislado y por fuera de la red corporativa [12], y

utilizando un solo usuario virtual, se encontró diferencias de un 30% en los tiempos de operación reportados. Este experimento pone de manifiesto el hecho de que las herramientas no coinciden entre sí. Entonces, ¿cómo se puede saber si coinciden en la realidad?

Las diferencias en los tiempos de operación las causan las diferentes formas en que las herramientas trabajan, y cómo se mide los tiempos de respuesta. Es posible tener una idea confiable sobre el rendimiento, de cualquiera de las herramientas, con base en las diferencias en los tiempos de operación bajo cargas diferentes. Los tiempos de operación absoluta pueden o no ser realistas, pero, por lo menos, el rendimiento relativo bajo cargas diferentes lo debe ser, y las mediciones reales lo respaldan. Además, las herramientas de monitoreo, como Perfmon, Rstatd, Tivoli, HPOpenview, por nombrar algunas, ayudan a determinar la carga que se debe colocar sobre una infraestructura determinada.

En la selección de la herramienta también se debe tener en cuenta la infraestructura de la aplicación bajo prueba. Si se utiliza un nivelador de carga en la infraestructura, sería muy útil poder emular el comportamiento de las diferentes direcciones IP que acceden al sistema, pero no todas las herramientas soportan esto. Además, si se tiene diferentes grupos de usuarios que ingresan a la aplicación a través de distintos canales (WAN, LAN y ADSL), se necesita la capacidad de emular el comportamiento de la infraestructura desde diferentes redes. Por lo general, este tipo de servicios solo se encuentra en herramientas comerciales [13].

Para la validación final es necesario conseguir algo de tiempo en el hardware de producción antes de entregar la aplicación bajo prueba, especialmente cuando difiere del entorno mismo. Para lograrlo es necesario incorporar el entorno de producción en el PDC, lo que no debe ser difícil de conseguir cuando no se utiliza el hardware de producción, por ejemplo, en un fin de semana. Un probador de desempeño siempre se preguntará por los datos que producen las herramientas y, de vez en cuando, interactúa manualmente con la aplicación bajo prueba para ver por sí mismo lo que experimentará el usuario cuando la aplicación esté en operación.

Una herramienta de automatización de pruebas, o una de pruebas de rendimiento, solo es una pequeña pieza de un *framework* de pruebas más grande. Puede ser difícil integrar una herramienta de código abierto con el software comercial en uso, pero, si es necesario este tipo de integración, entonces también debe ser parte de la PDC. Lo mismo ocurre con las herramientas comerciales, porque algunas requieren de otras de soporte de pruebas para hacerles seguimiento a los defectos, para controlar su procedencia y para administrar las pruebas, lo que obliga a que también se deban comprar; al final se termina pagando más de lo presupuestado.

Regularmente los proveedores de herramientas comerciales organizan PDC para mostrar lo que sus productos son capaces de hacer. Con el código abierto también existe quien esté dispuesto a ayudar, pero su objetivo es diferente, porque no quieren vender la herramienta debido a que probablemente es libre de todos modos, pero sí querrán vender servicios de consultoría. Si no existe esta ayuda se debe organizar la PDC de cuenta propia, aunque primero es necesario asegurar que la herramienta esté al alcance y que se ajusta a la situación específica.

3.6 Conocimientos y compromiso

Antes de comenzar a utilizar herramientas para automatizar pruebas, o para pruebas de rendimiento, se necesita cierto nivel de documentación de la prueba. Al mejorar la documentación se puede ganar algo de tiempo para la selección de la herramienta, lo que se reflejará en todo el proceso de automatización; este paso se debe realizar con independencia de las herramientas a

utilizar. Para la automatización de pruebas se debe contar con el estado de los resultados esperados en los casos de prueba y, aunque a menudo no están documentados, se espera que los probadores comprendan lo que pueden obtener. Estas herramientas no son inteligentes y se les debe indicar explícitamente lo que se espera como resultado. Para las pruebas de rendimiento se necesita saber lo que cada usuario está haciendo y cuándo lo está haciendo, es decir, se necesitan los perfiles de uso.

Otro inconveniente es la ausencia de un ingeniero de pruebas dedicado y experimentado. Sin importar cual herramienta se seleccione, se necesita tiempo y habilidad para implementarla y utilizarla y, si el equipo no cuenta con un ingeniero de este tipo para la automatización, el proyecto estará condenado al fracaso sin importar la herramienta que se utilice. Herramientas infalibles no existen, pero si se toma suficiente tiempo para pensar antes de iniciar, será posible elegir la más adecuada para el proyecto. Además, siempre se debe estar preparado para invertir en una herramienta complementaria. En la Tabla 2 se resume los pros y los contras de las herramientas de prueba de código abierto.

Tabla 2. Pros y contras de las herramientas de prueba de código abierto

Aspecto	Pros	Contras
Costos y licencias	Licencia inicial libre. Costos asociados bajos.	Puede requerir inversiones posteriores en asesorías y actualizaciones.
Plataformas y características	Plataformas y características limitadas al contexto.	Los usuarios pueden necesitar características que se adecuen a plataformas por fuera del contexto.
Probadores y desarrolladores	El trabajo de los probadores y los desarrolladores es colaborativo y en equipos integrales.	No es fácil conseguir que en los contextos de uso se puedan conformar equipos con estas características.
Objetos y protocolos	Se adaptan a los contextos específicos de uso.	Las modificaciones en objetos y protocolos generan nuevos desarrollos y modificaciones.
Herramientas e infraestructura	Son compatibles con la mayoría de herramientas de código abierto y se pueden instalar en todo tipo de infraestructura.	La compatibilidad puede disminuir cuando se trata de proyectos integradores de diferentes productos y en diferentes infraestructuras.
Conocimientos y compromiso	El mantenimiento y las actualizaciones al código las puede realizar el mismo usuario.	Se requieren amplios conocimientos de ingeniería para instalarlas, modificarlas y mantenerlas.

4. CÓMO EVALUAR HERRAMIENTAS DE CÓDIGO ABIERTO

Una pregunta frecuente de los usuarios potenciales del código abierto se relaciona con su fiabilidad. El código disponible para los productos de código abierto más populares y maduros tiende a ser revisado por cientos de desarrolladores, mucho más de los que la mayoría de empresas de software comercial puede tener para probar sus propios productos, por lo que lo que la calidad y la fiabilidad de este código tienden a ser altas.

Generalmente, el desarrollo de código abierto evita muchos de los procedimientos que normalmente se asumen como *buenas prácticas* en el desarrollo de software, pero sin afectar la calidad del producto. Por ejemplo, es poco probable que en proyectos de código abierto se documente detalles acerca de esfuerzos y planes de programación, plazos determinados y evaluaciones de riesgo. Pero sí se encontrará procesos de liberación rápidos e iterativos, que resultan de las mejoras continuas propuestas por los desarrolladores que contribuyen con iteraciones, mejoras y correcciones. Un estudio académico a 100 aplicaciones de código abierto [1] encontró que la calidad del código estructural resultaba ser mayor de lo esperado, que era comparable con el software comercial y que los proyectos más ampliamente conocidos, como

Apache y el kernel de Linux, en realidad muestran una densidad de defectos sustancialmente más bajos que los productos comerciales comparables. Esto hace posible que el código abierto ofrezca productos de alta calidad.

Sin embargo, ¿cómo separar lo bueno de lo malo? Con un poco de investigación en la web es fácil demostrar un número de áreas clave que respaldan la calidad del software de código abierto:

- *Una comunidad sostenible que desarrolla y depura código de forma rápida y eficaz*. Todos los proyectos de código abierto tienen áreas comunitarias en sus sitios, donde se puede ver el número de usuarios registrados y medir el volumen de actividad en los foros, *wikis* y páginas de las versiones, lo mismo que repositorios de código abierto legibles que permiten ver con qué frecuencia se entrega dicho código.
- *Documentación visible en el sitio del proyecto*. Un código modular soportado por buena documentación servirá para atraer a un nuevo cuerpo de desarrolladores y ampliar/atraer constructores. Como ejercicio, basta con preguntarle a un desarrollador comercial si es posible echarle un vistazo a su código para evaluar la calidad.
- *Un equipo central bien dirigido* que responde rápidamente a los comentarios de los revisores pares y a las contribuciones de código. Esto da como resultado una innovación rápida y a que se incremente la calidad, lo que es visible a través de los foros de los proyectos y de las listas públicas de correo.
- *Reutilización de librerías* de código establecido y probado, en lugar de escribir todo el programa desde el principio. Lo que ayuda a unificar una alta calidad, porque estas librerías deben ser obvias para la revisión de la documentación del desarrollador.

También existe algunos modelos formales que se pueden utilizar para lograr un marco de evaluación [13], como Navica's Open Source Maturity Model OSMM, que se publica bajo una licencia abierta y se utiliza para evaluar la madurez de los elementos clave del proyecto. El Business Readiness Rating es un modelo de evaluación desarrollado por organizaciones que buscan expandir los modelos OSMM para desarrollar un estándar abierto para el índice OSS.

5. CONCLUSIONES

El código abierto es un dominio amplio, y un enfoque reducido no puede conducir a una comprensión completa de todo el fenómeno; para lograrlo se debe realizar un análisis coherente de sus productos, de la forma en que se difunden y, lo más importante, de los efectos que su uso tiene sobre el dominio de TI. Sin importar cuál sea el beneficio que se logre siempre habrá problemas, los cuales se deben incluir en el portafolio de proyectos de los desarrolladores de software, cuya actividad se oriente a la eficiencia. Estos problemas se convierten en preguntas, procedimientos y soluciones que poco a poco se transformarán en productos de software libre de gran escala para uso general.

El área de problemas para la que se desarrolla herramientas de código abierto se ha convertido en fuerte competencia para las empresas que producen software licenciado. Bajo ninguna circunstancia los desarrolladores de software licenciado abren su código, aunque tengan los peores resultados reportados. Es una competencia tácita entre código abierto y software con licencia, en la que el primero es el motor principal.

La automatización de las pruebas da lugar a un proceso eficiente y ayuda a disminuir los costos de desarrollo y, aunque algunas pruebas todavía se tienen que realizar de forma manual, existe

algunas específicas en las que las herramientas automatizadas son inútiles. Al utilizar menos tiempo en la creación y aplicación de los casos de prueba, la automatización del proceso podría reducir los costos asociados, por lo que es importante seleccionar una herramienta adecuada para el éxito del proyecto.

Los productos de código abierto se desarrollan mediante trabajo colaborativo en el que el éxito se logra porque los miembros deben mostrar buena voluntad y responsabilidad. El carácter colaborativo, y esencialmente social, es necesario para hacerles frente a los grandes proyectos de código abierto. Una aplicación de este tipo crea un ambiente en el que las personas trabajan mejor juntas, pueden compartir información sin limitaciones de tiempo y espacio, y se caracteriza por aspectos como actividades conjuntas, medio ambiente compartido y forma de interacción. El desarrollo de código abierto no se organiza caóticamente, sino que sigue una dirección clara: incrementar el rendimiento para el procesamiento de grandes intereses.

El ritmo de desarrollo del código abierto está permanentemente en ascenso y el número de componentes distribuidos también es amplio; por todo esto es conveniente pensar en trabajos futuros como:

- Identificar los componentes de código abierto que los usuarios pueden utilizar en sus procesos.
- Crear, publicar y recomendar una jerarquía de componentes de código abierto de acuerdo con su calidad y comportamiento en uso.
- Integrar los componentes lo más rápidamente posible teniendo en cuenta las demandas del usuario. Cuando los componentes estén terminados es necesario disponer de motores que los encuentren e integren en una estructura unitaria en la que sean operativos. Todo instrumento debe estar destinado a crear una gestión avanzada en el campo del código abierto, porque un componente existe no necesariamente cuando se utiliza, sino cuando es escrito y publicado en una librería, y además debe ser usable, accesible y tener un procedimiento de fácil acceso. Si las listas de parámetros son demasiado largas en los datos de entrada de la prueba, se debe respetar unos requisitos estrictos, porque si los resultados no tienen una estructura flexible la aplicación se vuelve difícil de usar, y los usuarios se desaniman y la abandonan.

Se considera que el nivel de desarrollo alcanzado por la comunidad de código abierto es muy alto, y que la información es suficiente para caracterizarla completamente. De esta forma las entidades que la rigen y las técnicas y los métodos para gestionarla, muy pronto darán pie para el desarrollo de una nueva idea que cumpla con todos los requisitos necesarios para discutir acerca de los beneficios del código abierto en la prueba del software.

REFERENCIAS

[1] Ivan I. y Ciurea C. (2009). Quality characteristics of collaborative systems. En Second International Conference on Advances in Computer-Human Interactions. Cancun, Mexico.

[2] The Economist. (2009). Born free: Open-source software firms are flourishing, but are also becoming less distinctive. Recuperado: http://www.economist.com/node/13743278

[3] Henley M. y Kemp R. (2008). Open Source Software: An introduction. Computer Law & Security Report 24(1), 77-85.

[4] Hammond J. (2009). Open Source Software Goes Mainstream. Forrester Research.

[5] Crosman P. (2012). Did weak risk management controls worsen knight capital loss? Wall Street & Technology.

[6] Pocatilu P. (2002). Automated Software Testing Process. Economy Informatics 2(1), 97-99.

[7] Pocatilu P. (2006). Software Testing Costs. Economy Informatics 6(1), 90-93.

[8] Korel B. (1990). Automated Software Test Data Generation. IEEE Transactions on Software Engineering 16(8), 870-879.
[9] Ivan I. et al. (2007). Collaborative Systems Metrics. ASE Publishing House.
[10] Ciurea C. (2009). Collaborative Open Source Applications. Open Source Scientific Journal 1(1), 116-125.
[11] Vail C. (2005). Stress, load, volume, performance, benchmark and base line testing tool evaluation and comparison. Recuperado: http://www.vcaa.com/tools/loadtesttoolevaluationchart-023.pdf
[12] Ivan I. et al. (2007). Software Quality Verification through Empirical Testing. Journal of Applied Quantitative Methods 2(1), 38-60.
[13] Cristescu M. (2009). Open Source Software Reliability: Features and Tendence. Open Source Scientific Journal 1(1), 163-178.

CAPÍTULO XXV

Análisis a la eficiencia del conjunto de casos de prueba generados con la técnica *Requirements by contracts*[1]

Edgar Serna M.
Instituto Antioqueño de Investigación

En este capítulo se analiza el nivel de madurez del conocimiento y la eficiencia del conjunto de casos de prueba diseñados con la técnica *Requirements by Contracts* [1] para las pruebas funcionales, aplicada en un caso práctico. A partir del análisis del estado del arte es posible concluir que esta propuesta es incompleta para el objetivo de las pruebas, y que partiendo de la especificación funcional no es posible generar un conjunto de casos de prueba suficiente para ejecutar sobre el sistema bajo prueba.

[1] Presentado en el V Congreso Colombiano de Computación. Cartagena, Colombia. 2010.

INTRODUCCIÓN

La prueba es la última oportunidad en el proceso de desarrollo de software para detectar y corregir sus posibles defectos a un costo razonable, ya que la forma generalizada de trabajo utilizada por los profesionales en el área es la de ejecutarla sobre el producto *terminado* [2]; mientras que la práctica enseña que la prueba debe ser una actividad que se desarrolle de forma paralela a todo el proceso del ciclo de vida del producto [3]. Ya que es mucho más caro corregir los defectos que se detectan cuando el sistema se encuentra en operación [4], es importante poder confiar en que el conocimiento aplicado sea lo suficientemente formal como para obtener resultados predecibles en el proceso de las pruebas.

Las técnicas de prueba determinan diferentes criterios para diseñar los casos de prueba que se utilizarán como entradas para el sistema en estudio, lo que significa que un diseño efectivo y eficiente de ellos condiciona el éxito de las pruebas [5]. El conocimiento para seleccionar los métodos de prueba debe provenir de estudios que justifiquen los beneficios y condiciones de aplicación, sin embargo, los estudios formales y prácticos de este tipo no abundan, por lo que es difícil comparar los diferentes métodos de prueba (no tienen una sólida base teórica) y determinar qué variables de los mismos son de interés en estos estudios [6].

En vista de la importancia de contar con un conocimiento formal de las pruebas, en este capítulo se analiza el nivel de madurez del conocimiento en el área, y la eficiencia del conjunto de casos de prueba diseñado con la técnica *Requirements by Contracts* [1] para las pruebas funcionales. El objetivo principal es recoger el cuerpo de conocimiento acerca de los aspectos que esta propuesta tiene en cuenta para diseñar los casos de prueba y su nivel de madurez, de tal manera que esta información pueda ser útil a los desarrolladores para identificar las condiciones de aplicabilidad del método y la eficiencia del conjunto de casos de prueba que genera.

1. DESCRIPCIÓN DE LA TÉCNICA

La técnica se estructura en dos partes: en la primera se extienden los casos de uso utilizando contratos, en los que se incluye las pre y pos-condiciones y los casos de uso con sus parámetros; en la segunda se describe cómo generar automáticamente los casos de prueba a partir de los casos de uso extendidos. En un trabajo posterior de los mismos autores [7] se describe la aplicación de esta propuesta a familias de productos.

Parte de un diagrama de casos de uso en notación UML y, al final del proceso aplicado, se obtiene un modelo de casos de uso extendido con contratos y el conjunto de casos de prueba con los que se verifica la implementación del modelo. Los casos de uso se expresan mediante caminos de ejecución, que recorren las secuencias de los casos de uso para satisfacer las pre y pos-condiciones.

1.1 Pasos de la propuesta

En la Figura 1 se presenta gráficamente la secuencia de pasos que aplica la propuesta y en la Tabla 1 el resumen de su descripción.

- Paso 1. Se utiliza un intérprete (lenguaje) de contratos para extender los casos de uso. El lenguaje permite incluir los parámetros y las pre y pos-condiciones, expresadas como proposiciones. El objetivo es formular las dependencias que existen entre los casos de uso.

- Paso 2. Se construye y ejecuta un modelo de ejecución de casos de uso. Consiste en realizar un diagrama que permita expresar, a partir de los casos de uso extendidos, el comportamiento del sistema. En este diagrama los nodos representan estados del sistema, determinados por las proposiciones, y las transiciones instancias del caso de uso. No existe restricciones para definir el significado semántico de los predicados.
- Paso 3. Se selecciona el criterio de cobertura. La propuesta propone cuatro criterios para recorrer el modelo de ejecución, determinar el criterio de cobertura y obtener los casos de prueba: 1) de todos los bordes: por lo menos existe un objetivo de prueba por transición; 2) de todos los vértices: por lo menos existe un objetivo de prueba por vértice; 3) de todos los casos de uso instanciados: por lo menos existe un objetivo de prueba por caso de uso instanciado; y 4) de todos los vértices y casos de uso instanciados: por lo menos existe un objetivo de prueba por vértice y caso de uso instanciado.
- Paso 4. Se aplica el criterio de cobertura seleccionado. Al final se obtiene un conjunto de instancias de casos de uso con sus parámetros, conformado por secuencias de casos de uso válidas.
- Paso 5. Mediante una herramienta para generar pruebas y a partir del conjunto de instancias del paso anterior se genera los casos de prueba.

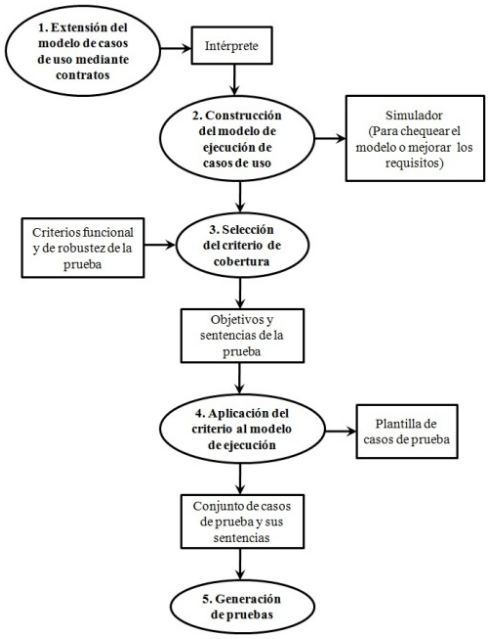

Figura 1. Diagrama de actividades de la propuesta

Tabla 1. Resumen de los pasos de la propuesta

	Detalle	Objetivo
1	Se extiende el modelo de casos de uso utilizando un lenguaje de contratos	Obtener casos de uso con parámetros, y pre y pos-condiciones
2	Se construye el modelo de ejecución de casos de uso	Diseñar el modelo de ejecución de casos de uso
3	Se selecciona el criterio de cobertura	Describir el criterio de cobertura para recorrer el modelo construido
4	Se aplica el criterio de cobertura seleccionado	Obtener secuencias de casos de uso válidas
5	Se generan los casos de prueba	Diseñar las pruebas ejecutables para el sistema

2. CASO APLICATIVO

A continuación, se describe el proceso utilizado para aplicar experimentalmente esta técnica.

2.1 Los casos de uso

El sistema sobre el que se experimenta la propuesta es un caso netamente académico. Para una situación industrial y con exigencia más compleja, la aplicación tiene un nivel de trabajo y complejidad un poco mayor. El sistema utilizado para experimentar la propuesta se basa en la circulación de material en una biblioteca; en la Tabla 2 se detalla los casos de uso utilizados y en La Figura 2 se presenta el diagrama de casos de uso de aplicación.

Tabla 2. Casos de uso seleccionados del sistema de circulación de material

Referencia	Descripción
CU-001	Validar Usuario
CU-002	Prestar material
CU-003	Devolver material

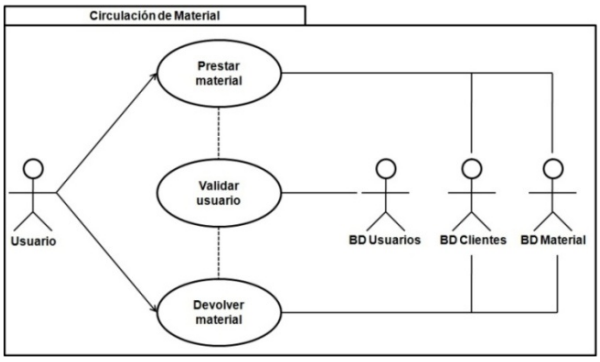

Figura 2. Diagrama de casos de uso del sistema

En las tablas 3, 4 y 5 se describe la documentación de los casos de uso mediante la aplicación de la plantilla sugerida en [8]. Para que la documentación no se extienda innecesariamente se suprime algunos elementos del modelo de la plantilla no necesarios en este caso aplicativo.

Tabla 3. Caso de uso *Validar Usuario*

Caso de uso	CU-001 Validad Usuario
Actores	Usuario, BD usuarios
Tipo	Inclusión
Propósito	Permitir la validación de usuarios en el sistema
Resumen	El usuario carga la página inicial del sistema y digita nombre de usuario y contraseña
Pre-condiciones	Ninguna
Flujo Principal	1. El usuario carga la página de acceso al sistema 2. El sistema despliega la página de verificación de datos de ingreso y solicita usuario y contraseña 3. El usuario digita los datos y selecciona la opción *Ingresar* 4. El sistema valida usuario y contraseña, y despliega la página inicial del proceso
Excepciones	E1: El sistema no carga la página de acceso si el servidor no está activo o existen problemas de navegación, entonces se termina el caso de uso y muestra un mensaje de error E2: Si los datos de usuario y contraseña no se digitan, el sistema los solicita nuevamente y despliega un mensaje de error E3: Si el nombre de usuario no se encuentra en la base de datos de usuarios, el sistema lo solicita nuevamente y despliega un mensaje de error E4: Si la contraseña no es válida, el sistema la solicita nuevamente y despliega un mensaje de error

Tabla 4. Caso de uso *Prestar Material*

Caso de uso	CU-002 Prestar Material
Actores	Usuario, BD Material, BD Clientes
Tipo	Principal
Propósito	Registrar el préstamo del material de la biblioteca
Resumen	El usuario atiende las solicitudes de los clientes y las registra o rechaza de acuerdo con el escenario
Pre-condiciones	El usuario debe estar validado en el sistema
Flujo Principal	1. El sistema presenta el cuadro de diálogo de préstamo de material 2. El usuario digita el código del material que el cliente desea prestar 3. El sistema valida la disponibilidad del material y el estado del cliente, y despliega un mensaje de confirmación
Excepciones	E1: Si no se digita el código de material el sistema lo solicita nuevamente y despliega un mensaje de error E2: Si el código no existe despliega un mensaje de error y termina el caso de uso E3: Si el código no está disponible se despliega un mensaje de error y termina el caso de uso E4: Si el cliente está deshabilitado se despliega un mensaje de error y termina el caso de uso

Tabla 5. Caso de uso *Devolver material*

Caso de uso	CU-003 Devolver Material
Actores	Usuario, BD Material, BD Clientes
Tipo	Principal
Propósito	Registrar la devolución de material a la biblioteca por los clientes
Resumen	El usuario atiende las devoluciones de material por los clientes
Pre-condiciones	El usuario debe estar validado en el sistema
Flujo Principal	1. El sistema presenta el cuadro de diálogo de devolución de material 2. El sistema despliega la información del material que el cliente tiene en préstamo 3. El usuario selecciona el código a devolver 4. El sistema actualiza las bases de datos de Clientes y Material 5. El sistema despliega mensaje de confirmación de devolución
Excepciones	E1: Si el cliente no tiene material en préstamo el sistema muestra el mensaje y termina el caso de uso

2.2 Generar el conjunto de casos de prueba

En el diagrama de casos de uso se aprecia que existe dependencia de los casos de uso *Prestar material* y *Devolver material* con el de *Validar usuario*, de tal manera que no pueden ejecutarse antes de éste; de igual manera existe una dependencia temporal entre los dos primeros, ya que prestar debe haberse ejecutado antes que devolver. A continuación, se aplica la propuesta a estos casos de uso para obtener el conjunto de casos de prueba del sistema.

- *Paso 1*. Ampliar el diagrama de casos de uso mediante contratos. El resultado aplicar este paso se aprecia en la Tabla 6.

Tabla 6. Contratos para los casos de uso del sistema

CU	Función	Pre-condición	Pos-condición
001	Validar(u1)	Ninguna	UsuarioValidado (uv)
002	Prestar (usuario uv)	UsuarioValidado (usuario uv)	MaterialPrestado (mp)
003	Devolver (usuario uv, mp)	UsuarioValidado (usuario uv) y MaterialPrestado (mp)	MaterialDevuelto (md)

- *Paso 2*. Modelo de ejecución de los casos de uso. El resultado aplicar este paso se aprecia en la Tabla 7.

Tabla 7. Descripción de la semántica de los predicados para ejecución de los casos de uso

Predicado	Detalle
Validar(u1)	Determina que el usuario u1 debe quedar validado (uv) luego de ejecutar el caso de uso *Validar Usuario*
MaterialPrestado(mp)	Determina que el material mp se encuentra en estado préstamo
MaterialDevuelto(md)	Determina que el material md se encuentra en estado devuelto y puede prestarse nuevamente

De los parámetros y las pre y pos-condiciones expresadas en los contratos de la Tabla 6 se observa el orden de ejecución de los casos de uso (Figura 3). El modelo de ejecución detalla el progreso de los diferentes parámetros de los casos de uso; los estados detallan el estado del sistema de acuerdo con los predicados definidos en los contratos, y cada transición representa la ejecución de un caso de uso.

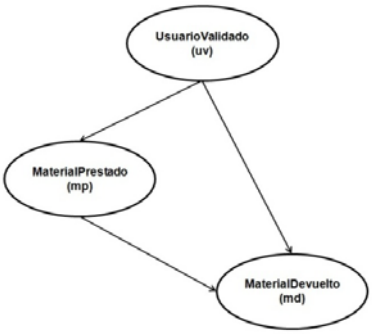

Figura 3. Modelo de ejecución de los casos de uso

- *Paso 3*. Selección del criterio de cobertura. En este caso aplicativo se selecciona el criterio de cobertura de todos los estados y transiciones, por lo que todos los del modelo de la Figura 3 se deben considerar en, por lo menos, una prueba.

- *Paso 5*. Generar el conjunto de casos de prueba. Para ejecutar este paso se utiliza la herramienta de libre distribución disponible en [9]. Para utilizar esta herramienta es necesario describir el modelo de ejecución de la Figura 4.

```
# Entidades del sistema
{
   u1 : usuario
   md1, md2 : MaterialDisponible
}

# Descripción del estado inicial
{
   MaterialDisponible (md1)
   MateriaDisponible(md2)
}

# Descripción de los casos de uso

# Caso de uso ValidarUsuario
CU Validar(u1: usuario)
pre
post UsuarioValidado(uv)
```

```
# Caso de uso PrestarMaterial
CU Prestar(uv: UsuarioValidado; md : MaterialDisponible)
pre UsuarioValidado(uv) y MaterialDisponible(md)
post prestamo(L) and not disponible(L)

# Caso de uso DevolverMaterial
CU Devolver(uv : UsuarioValidado; mp : MaterialPrestado)
pre usuarioValidado(uv) y MaterialPrestado(mp)
post MaterialDevuelto(md) y MaterialDisponible(md)
```

Figura 4. Descripción del Modelo de ejecución

3. RESULTADOS

En las Figuras 5 a 8 se representa las secuencias que surgen al aplicar la herramienta descrita al modelo de ejecución, con todos los criterios de cobertura.

```
[ValidarUsuario(u1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv,md1), Devolver(uv,mp1)]
[Validadusuario(u1), Prestar(uv, md2), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md2), Prestar(uv, md1)]
[ValidarUsuario(u1), Prestar(uv, md2), Devolver(uv, mp2)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2),
ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2),
Devolver(uv, mp1)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2),
Devolver(uv, mp2)]
```

Figura 5. Criterio de todos los bordes

```
[ValidarUsuario(u1), Prestar(uv, md2)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2)]
```

Figura 6. Criterio de todos los vértices

```
[ValidarUsuario(u1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), Devolver(uv,mp1)]
[ValidarUsuario(u1), Prestar(u1, md2), ValidarUsuario(u1)]
[Validarusuario(u1), Prestar(uv, md2), Devolver(uv, mp2)]

Lo que se debe cubrir:
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2),
Devolver(uv,mp1), Devolver(uv, mp2)]
```

Figura 7. Criterio de todos los casos de uso instanciados

```
[ValidarUsuario(u1), Validarusuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), Devolver(uv, mp1)]
[ValidarUsuario(u1), Prestar(uv, md2), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md2), Devolver(uv, mp2)]
[ValidarUsuario(u1), Prestar(uv, mp2)]
[Validarusuario(u1), Prestar(uv, md1), Prestar(uv, md2)]

Lo que se debe cubrir:
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2),
Devolver(uv,md1), Devolver(uv, mp2)] : 5
```

Figura 8. Criterio de todos los vértices y casos de uso instanciados

La herramienta no permite la ejecución del proceso en el criterio de casos de uso instanciados, ni en el de vértices y casos de uso instanciados, porque presenta un error de bloqueo interno y se detiene el proceso de generación de los casos de prueba.

3.1 Factores de análisis

Para lograr una visión más amplia de la técnica se establece una serie de características elegidas como las que mejor describen la mayoría de sus cualidades, y facilitan su validación y comparación en pro de analizar la eficiencia del conjunto de casos de prueba, de tal forma que al leerlas es posible conocer lo más importante de la propuesta y los diferentes tratamientos dados a cada una de ellas. Los factores que en este estudio se tienen en cuenta para evaluar la eficiencia de la propuesta fueron promulgados en [10]. Los resultados se resumen en la Tabla 8.

1. Origen: requisitos para aplicación. Tipo: *Set{Necesidades, lenguaje natural, casos de uso, varios}*
2. Notación: si introduce nuevas notaciones gráficas. Tipo: *Boolean.*
3. Aplicación: si referencia proyectos reales de aplicación. Tipo: *Boolean*.
4. Estándares: si utiliza diagramas y notaciones estándar o comúnmente aceptados. Tipo: *Boolean*.
5. Herramientas: si existen herramientas que la soporten. Tipo: *Boolean*.
6. Prácticos: si incluye casos prácticos y ejemplos de aplicación. Tipo: *Boolean*.
7. Cobertura: si para generar los casos de prueba describe criterios de cobertura. Tipo: *Set{Enum{Análisis de caminos, partición de categorías, varios, no}}*.
8. Valores para prueba: si detalla cómo seleccionar los valores para el conjunto de casos de prueba generados. Tipo: *Boolean*.
9. Optimización: si describe cómo seleccionar subconjuntos de casos de prueba sin perder cobertura de requisitos. Tipo: *Boolean*.
10. Dependencia: si genera casos de prueba que dependan de varios casos de uso o de casos de uso aislados. Tipo: *Boolean*.
11. Orden de ejecución: si describe el orden de ejecución de los casos de prueba que genera. Tipo: *Boolean*.
12. Construcción modelo: si incluye la descripción de cómo construir un modelo de comportamiento del sistema o si genera los casos de pruebas desde los datos de los casos de uso. Tipo: *Boolean*.
13. Priorización: si genera pruebas más detalladas para los requisitos o casos de uso primarios y menos detalladas los secundarios. Tipo: *Boolean*.
14. Continuidad: si el o los autores continúan trabajando en la propuesta, si están desarrollando herramientas de soporte, si han ofrecido nuevas versiones o si han recibido el apoyo de otros grupos de trabajo. Tipo: *Enum{Sí, no, ?}*.
15. Indicador de inicio: si indica el momento para iniciar la generación de casos de prueba. Tipo: *Enum{Elicitación de requisitos, Análisis del sistema}*.
16. Resultados: cómo se presentan los resultados de aplicación. Tipo: *Set{Enum{Pruebas en lenguaje no formal, scripts de prueba. Modelo de prueba, secuencia de transiciones}}*.
17. Documentación: cuál es el grado dificultad para aplicarla de acuerdo a la calidad de la documentación ofrecida. Tipo: *Set{Alta, media, baja}*. Alta, si la documentación es suficiente; media si existen lagunas o vacíos que la documentación no resuelve; baja, si es difícil o no logra implementar con la documentación.
18. Pasos: cantidad de pasos propuestos. Tipo: *Entero*.
19. Año publicación: para seguimiento más detallado. Tipo: *Entero*.

Tabla 8. Resultado de las características

Característica	Indicador
1. Origen	Casos de uso
2. Notación	Diagramas de transición de casos de uso
3. Aplicación	No
4. Estándares	Sí
5. Herramientas	Sí
6. Prácticos	Sí
7. Cobertura	Varios
8. Valores para prueba	No
9. Optimización	Sí
10. Dependencia	Sí
11. Orden de ejecución	Sí
12. Construcción modelo	Sí
13. Priorización	No
14. Continuidad	Sí
15. Indicador de inicio	Elicitación de requisitos
16. Resultados	Secuencias de transiciones entre los casos de uso
17. Documentación	Media
18. Pasos	5
19. Año publicación	2003/2004

3.2 Limitaciones del estudio

Debido a que este estudio de caso es académico, es posible que los resultados no tengan un nivel de representatividad como los que podrían hallarse en un caso industrial. Otra limitante lo constituye la herramienta para automatizar la propuesta, ya que su error no se pudo subsanar, por lo que es posible que los casos de prueba obtenidos no sean los más adecuados para analizar su eficiencia. Por último, el experimento puede estar limitado al momento de seleccionar los factores de análisis, porque si el estudio de caso es de corte industrial, es posible que sean otros los que hay que tener en cuenta.

4. ANÁLISIS DE RESULTADOS

- No es suficiente para ponerla en práctica la forma como describe el proceso para la generación del diagrama de comportamiento del sistema.
- La notación que ofrece para el diagrama de transición de casos de uso es incompleta en relación con los diagramas de estados de UML. Por ejemplo, no detalla qué hacer cuando no se tiene un estado, como el caso del estado inicial.
- La utilidad del diagrama de comportamiento no es clara, ya que el modelo de comportamiento de la herramienta de soporte se puede generar desde el diagrama de casos de uso extendido con los contratos.
- La herramienta de soporte no está bien documentada, tal es el caso del formato de los archivos que utiliza, por lo que debe recurrirse a la información en la Internet para utilizarlos.
- No describe como pasar del modelo de comportamiento a la descripción de la herramienta, por lo que es necesario intuirlo desde los ejemplos y casos prácticos disponibles.
- La herramienta de soporte, como los autores lo especifican, está en un estado incipiente de desarrollo.
- En algunos criterios se encontraron errores no documentados y no se pudo determinar su origen.

- La cantidad de casos de prueba varía mucho de acuerdo con el criterio de cobertura.
- Es bueno que la especificación que toma como punto de partida sea el UML, dado su nivel de unificación en el campo de conocimiento.
- Trata de forma tangencial el concepto de diseño por contratos, una idea del desarrollo que es posible aplicar a las pruebas para buscar, en algún momento, que sea posible formalizarlas.

4.1 Puntos fuertes y débiles de la propuesta

Sus puntos fuertes los constituyen: contar con diferentes criterios de cobertura para recorrer el modelo de ejecución; contar con una herramienta preliminar que la soporte, el UCTSystem [9]; permite generar pruebas en las que se involucra las secuencias de casos de uso. Entre sus puntos débiles se encuentra el no permitir que se desarrolle pruebas para verificar aisladamente el comportamiento de cada caso de uso. En la Tabla 9 se resumen la fortalezas y debilidades de la técnica *Requirements by contract*.

Tabla 9. Resumen de fortalezas y debilidades

Fortalezas	Debilidades
Contar con una herramienta de libre distribución en la que se implementan los criterios que propone	Extender los casos de uso de forma no estandarizada por UML
Generar pruebas que se basan en las secuencias de los casos de uso	No permitir que se desarrollen pruebas para verificar aisladamente el comportamiento de cada caso de uso
Utilizar contratos para expresar dependencias de los casos de uso entre sí	No detalla con qué criterio se selecciona el número de parámetros para un caso de uso
	No hay forma de relacionar los parámetros simbólicos con los valores reales
	Muchos de los conceptos que utiliza no tienen una definición o descripción suficiente
	No tiene una descripción detallada de cómo implementar las pruebas
	No detalla estudios de caso reales o industriales en los que se haya aplicado con éxito

5. CONCLUSIONES

A partir del análisis del estado del arte es posible concluir que esta técnica está incompleta, y que partiendo de la especificación funcional no es posible generar un conjunto de casos de prueba suficiente para ejecutar sobre el sistema bajo prueba.

De la descripción de esta propuesta y del análisis presentado en la Tabla 9, aunque presenta algunos puntos fuertes, sus carencias hacen que sea difícil ponerla en práctica.

El estudio que realiza al problema por resolver es incompleto, ya que no tiene en cuenta elementos que puedan considerarse fundamentales, como generar valores concretos de prueba o evaluar la cobertura de los requisitos.

La documentación que acompaña la propuesta es de calidad media, en su aplicación aparecieron escenarios o dudas que no fue posible resolver; lo que le resta eficiencia ya que esas situaciones se deben resolver desde la experiencia, la intuición o el conocimiento de quien que la aplique.

Le hace falta consistencia, ya que en su presentación describe que es posible obtener directamente el conjunto de casos de prueba: valores de prueba, interacciones con el sistema y

resultados esperados; pero en la práctica no fue posible obtener pruebas directamente ejecutables ni generar los resultados esperados; a cambio se obtuvo un conjunto de tablas en formato propio.

No incluye cómo evaluar la calidad del conjunto de casos de prueba generados. Parte del concepto de que es suficiente con seguir adecuadamente sus pasos para obtener un conjunto de casos de prueba de calidad y máxima cobertura.

La automatización del proceso de generación del conjunto de casos de prueba no es completamente posible con esta herramienta, ya que los requisitos se describen en lenguaje natural.

Aunque sigue los estándares UML su aplicación no permite un seguimiento al 100% de los mismos.

No detalla cómo resolver los ciclos infinitos que aparecen en el código del programa en prueba. Para el caso del sistema detallado en este documento, cuando un usuario en el caso de uso *Validar Usuario* digita el nombre o la contraseña equivocada y el sistema los vuelve a solicitar, la propuesta no restringe el número de intentos y lo deja a criterio de los desarrolladores, lo que dificulta su automatización.

Para complementarla se propone como trabajo futuro:

- Es necesario que la propuesta se pueda conectar a una herramienta CASE UML, utilizando una herramienta para generar el conjunto de casos de prueba, más específica y adecuada, y menos *ad hoc*.
- Validar la propuesta en otros estudios de caso. El aquí presentado y el que los autores utilizan para documentarla, tienen un corte muy académico, por lo que es conveniente una aplicación a procesos industriales más exigentes.
- Debe estudiarse las variantes funcionales para productos de software en línea y la flexibilidad de la propuesta, ya que frecuentemente los requisitos se modifican en el ciclo de vida.
- Se requiere un campo de conocimiento más estandarizado para determinar los criterios utilizados para calificar la eficiencia de las propuestas recientes. Los que se encuentra no contemplan algunas características que dichas propuestas utilizan en sus procesos para obtener los casos de prueba, y que han aparecido igualmente de forma reciente.

REFERENCIAS

[1] Nebut C. et al. (2003). Requirements by Contracts allow Automated System Testing. En 14th International Symposium on Software Reliability Engineering. Denver, USA.

[2] Kamde P. et al. (2006). Value of test cases in software testing. Management of Innovation and Technology 2, 668-672.

[3] Leon D. et al. (2007). An empirical evaluation of test case filtering techniques based on exercising complex information flows. IEEE Transactions on Software Engineering 33(7), 454-477.

[4] Lewis W. (2000). Software testing and continuous quality improvement. CRC Press.

[5] Sinha P. y Suri N. (1999). Identification of test cases using a formal approach. En Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing. Madison, USA.

[6] Pressman R. (2005). Software engineering: a practitioner's approach. McGraw Hill.

[7] Nebut C. et al. (2004). A Requirement-Based Approach to Test Product Families. Lecture Notes in Computer Science 3014, 198-210.

[8] Weitzenfeld A. (2005). Ingeniería de Software Orientada a Objetos con UML, Java e Internet. Thompson.
[9] Nebut C. y Fleurey F. (2003). UCTSystem: Generating tests from requirements. En 14th International Symposium on Software Reliability Engineering. Denver, USA.
[10] Bach J. (1995). The Challenge of Good Enough Software. American Programmer. Yourdon Press.

CAPÍTULO XVI
Análisis crítico a las propuestas para generar casos de prueba desde los casos de uso para las pruebas funcionales[1]

Edgar Serna M.[1]
Fernando Arango I.[2]
[1]Instituto Antioqueño de Investigación
[2]Universidad Nacional de Colombia

El hecho de formalizar su conocimiento les permite a las disciplinas ingenieriles lograr resultados predecibles. Lamentablemente, el conocimiento utilizado en la Ingeniería del Software puede considerarse de un nivel de madurez relativamente bajo; los desarrolladores se guían por la intuición, la moda o lo que dicta el mercado, en lugar de los hechos o declaraciones indiscutibles propias de una disciplina ingenieril. Las propuestas de pruebas determinan los diferentes criterios para diseñar los casos de prueba, que se utilizan como entradas para examinar un sistema objeto de estudio; lo que significa que diseñar eficaz y eficientemente los casos de prueba es una condición de éxito para las pruebas. El conocimiento que permita seleccionar un método de prueba y un conjunto de casos de prueba, debe surgir de estudios que justifiquen los beneficios y las condiciones de aplicación de los mismos. En este trabajo se analiza el nivel de madurez del conocimiento acerca de los métodos de diseño de los casos de prueba, generados para la prueba funcional mediante un análisis crítico de las propuestas desarrolladas en esta temática.

[1] Publicado en la Revista Avances en Sistemas e Informática 7(2), 105-113. 2010.

INTRODUCCIÓN

El término *prueba de software* comprende un conjunto de métodos, técnicas y conocimiento cuyo objetivo es determinar la calidad del software mediante el análisis a su funcionamiento.

Los métodos de prueba proporcionan diferentes criterios para diseñar el conjunto de casos de prueba que se utilizarán para probar el software, y que permiten agrupar los métodos en familias. De esta manera, los métodos que pertenecen a la misma familia son similares en lo que respecta a la información que necesitan para diseñar los casos de prueba (código fuente o especificaciones) o el cómo se aplican los casos de prueba (flujos de control, flujos de datos, errores típicos, etc.).

El objetivo de este estudio no es describir las características de los métodos de prueba o de sus familias, ya que esta información puede ser obtenida de la literatura clásica relacionada, por ejemplo, Beizer [1] y Myers [2, 3]; en cambio, se centra en la descripción y el detalle de las propuestas utilizadas para diseñar casos de prueba desde el enfoque de las pruebas funcionales.

En vista de la importancia de contar con un conocimiento formal de las pruebas, en este documento se analiza el nivel de madurez del conocimiento en el área y de los casos de prueba generados para las pruebas funcionales. Para tal propósito se hace un análisis crítico de algunas de las más importantes propuestas para generar casos de prueba a partir de los requisitos funcionales de los sistemas.

El objetivo principal es recoger el cuerpo de conocimiento acerca de los aspectos que las propuestas tienen en cuenta para diseñar los casos de prueba y su nivel de madurez, de tal manera que esta información pueda ser útil a los desarrolladores para identificar las condiciones de aplicabilidad de los diferentes métodos y los casos de prueba que generan.

1. LAS PRUEBAS FUNCIONALES

La familia de las pruebas funcionales propone un enfoque en el que la especificación del programa se utiliza para diseñar los casos de prueba. El componente a probar se considera como una caja negra cuyo funcionamiento se determina al estudiar las entradas y las salidas asociadas.

Del conjunto de posibles entradas del sistema, esta familia considera el sub-conjunto formado por las entradas que hacen que funcione de forma anormal; y la clave para diseñar los casos de prueba consiste en encontrar las entradas que tienen una alta probabilidad de pertenecer a este sub-conjunto.

Para tal propósito el método divide las entradas al sistema en sub-conjuntos, denominados clases de equivalencia, donde cada elemento que la conforma se comporta de manera similar, a fin de que todos los elementos en ella sean las entradas que causen tanto el funcionamiento normal como el anormal del sistema. Los métodos que conforman esta familia difieren entre sí en cuanto a la rigurosidad con la que cubren las clases de datos seleccionados.

2. ANÁLISIS A LAS PROPUESTAS IDENTIFICADAS

Las propuestas analizadas en este estudio no son las únicas que se ha promulgado para diseñar casos de prueba desde los casos de uso para las pruebas funcionales, la selección para el análisis se hizo de aquellas propuestas que cumplieran con las características de haberse promulgado en el siglo XXI, y que tuvieran como punto de partida los requisitos funcionales del software especificados en casos de uso.

2.1 Automated test case generation from dynamic models [4]

La propuesta parte de un caso de uso descrito en lenguaje natural y anotado en una plantilla recomendada [5]; se estructura en dos bloques: en el primero se realiza la traducción de los casos de uso a diagramas de estados, es decir, se traducen desde el lenguaje natural al diagrama; en la propuesta se incluye un resumen de las reglas que se debe aplicar para generar dicho diagrama desde la definición de los casos de uso [6].

En el segundo bloque, a partir de los diagramas de estados del bloque anterior, se realiza las siguientes actividades: 1) se toman las pre y pos-condiciones del diagrama y se traducen a proposiciones conformadas por un identificador para cada una de ellas; 2) dado que puede existir proposiciones que no dependen de los estados ni transiciones del diagrama, es necesario hacer una extensión a la actividad anterior, se anexa una proposición en la que el conjunto de datos es válido (está definido), y otra en la que no es válido (no está definido); 3) sobre el conjunto de proposiciones extendidas se genera las operaciones, proceso que consiste en una traducción sistemática utilizando técnicas de inteligencia artificial; 4) se especifica los estados inicial y final del conjunto resultante de proposiciones (conjunto de requisitos, adicciones y sustracciones); 5) se define con qué criterio se aplicará la cobertura de la prueba mediante un programa de inteligencia artificial, que toma las transiciones del diagrama de estados y desde su estado inicial analiza el posible estado final; y 6) mediante la aplicación de un algoritmo se traduce el diagrama de estados al lenguaje STRIPS [7], y se genera el conjunto de casos de prueba; el algoritmo permite hallar el conjunto de operaciones que, desde las pre-condiciones, obtienen las pos-condiciones.

El producto final de esta propuesta es un conjunto de transiciones posibles en el diagrama de estados expresadas con operadores, y las proposiciones iniciales y finales del mismo. Presenta una explicación detallada del proceso de generación de pruebas, que ilustra descriptivamente; puede aplicarse con cualquier herramienta que soporte STRIPS y describe claramente cómo establecer la cobertura.

No explica cómo se extrae las proposiciones del diagrama de estados; no tiene en cuenta las dependencias de los casos de uso; trata los casos de uso de manera muy aislada; no automatiza la traducción del caso de uso al diagrama de estados; en la medida que se incrementa la complejidad de los datos también se incrementa las proposiciones y operaciones; el hecho de expresar las pruebas con proposiciones y operaciones dificulta la implementación, ya que es necesario realizar retrocesos para describirlas.

2.2 Boundary Value Analysis [8]

Es una propuesta que selecciona los datos de prueba de aquellos cuyo valor está a lo largo de sus límites, es decir, selecciona los datos de las fronteras superior e inferior del valor a probar. Incluye los valores máximo, mínimo, justo dentro y justo fuera de los límites, los valores característicos y los valores de error. La idea es que, si un sistema funciona correctamente para estos valores, funcionará correctamente para todos los valores entre ellos [9]. Tradicionalmente comienza por identificar el incremento de valor más pequeño en una categoría específica de equivalencia; este incremento se llama el valor límite de *épsilon*, y se utiliza para calcular los valores máximos y mínimos en torno a una clase de equivalencia.

Los pasos para utilizar la prueba de valores límite son simples: primero se identifican las clases de equivalencia; luego se identifican los límites de cada clase, y posteriormente se diseña los casos de prueba para cada valor límite y mediante la selección de un punto de la frontera, un punto

justo debajo y un punto justo por encima [10]. *Debajo* y *encima* son términos relativos que dependen de las unidades de valor de los datos. Se debe tener en cuenta que un punto justo debajo o encima de un límite puede estar en otra clase de equivalencia, por lo que no hay ninguna razón para repetir la prueba.

Esta propuesta reduce significativamente el número de casos de prueba que se crean y ejecutan; es más apropiada para sistemas en los que gran parte de la toma de datos para valores de entrada está dentro de rangos o en conjuntos; es aplicable en las pruebas de unidad, de integración, de sistema y de aceptación; todo lo que requiere son entradas que puedan ser particionadas, y fronteras que puedan ser identificadas con base en los requisitos funcionales del sistema; existe documentación suficiente, ejemplos y casos de éxito que la soportan [11-16].

Por otro lado, la herramienta de aplicación es demasiado complicada para utilizar y no ofrece una ayuda clara; su dependencia de otras técnicas, como la de clases de equivalencia, reduce su homogeneidad e independencia, ya que el probador debe conocerlas también.

2.3 Test cases from use cases [17]

Parte del principio de que las pruebas se deben diseñar desde las primeras etapas del ciclo de vida del producto, y describe cómo utilizar los casos de uso en la generación de los casos de prueba. El caso de uso se define textualmente en lenguaje natural y en una plantilla. Consiste en: 1) generar los escenarios de prueba de los casos de uso, donde se identifica todas las combinaciones posibles entre la ruta principal de ejecución y las alternas, y se enuncian en una tabla; 2) identificar el conjunto de casos de prueba (conjunto de entradas, condiciones de ejecución y resultados esperados) para cada uno de los escenarios y condiciones de ejecución; esta información también se enuncia en tablas, pero sin notación o formalismo; y 3) identificar el conjunto de valores para cada caso de prueba. Al final del proceso el resultado es una tabla en la que se describe, en lenguaje natural, todos los casos de prueba que permitan verificar que la implantación del caso de uso es correcta.

Aunque no indica un modelo formal para presentar el caso de uso, describe los elementos que debe contener; tampoco indica cómo se obtiene los valores de los datos para el tercer paso; es una propuesta sencilla y simple de aplicar, pero le falta detalle y rigor en la descripción; ofrece poca escalabilidad para procesos más complejos; debido a que trata los casos de uso aisladamente, no es posible observar la dependencia entre ellos; el lenguaje natural en el que está expresada no facilita su automatización; el resultado de aplicarla a casos de uso complejos es un elevado número de casos de prueba; aunque parte del principio de diseñar los casos de prueba desde el comienzo del proyecto, no explica cómo hacerlo; y no describe las reglas sistemáticas que permitan aplicar los pasos.

2.4 Requirement Base Testing [18-20]

Parte del conjunto de requisitos en lenguaje natural y, mediante la aplicación de dos bloques de actividades, genera los casos de prueba. Como resultado se obtiene un conjunto de casos de prueba expresados en lenguaje natural y estructurado en un modelo causa efecto o estado esperado. El primer bloque, revisión de requisitos, está dividido en cuatro actividades: 1) se analiza los objetivos del sistema y se validan con los requisitos; 2) a esos requisitos se les aplica los casos de uso; 3) se hace una revisión no detallada de ambigüedades, que consiste en eliminar todas las palabras y frases ambiguas de la descripción de los requisitos; 4) los integrantes del equipo más conocedores del dominio del sistema revisan la descripción resultante hasta el momento.

El segundo bloque, generación y revisión de casos de prueba, conlleva la realización de ocho actividades: 1) de la descripción de los requisitos se genera los diagramas causa efecto, que luego se traducen a tablas de decisión en las que se incluye causas, efectos y posibles combinaciones; cada conjunto de combinaciones se constituye en un potencial caso de prueba; 2) se hace una verificación de consistencia del proceso hasta el momento; 3) los ingenieros de requisitos hacen una revisión de los casos de prueba generados; 4) los casos de prueba son revisados por los usuarios; 5) los desarrolladores revisan los casos de prueba; 6) se revisa los casos de prueba sobre el modelo del diseño del sistema; 7) las casos de prueba son revisados sobre el código; 8) se ejecuta los casos de prueba.

La propuesta, aunque extensa, no ofrece una documentación adecuada acerca de los productos que se generan en cada actividad, lo que imposibilita su aplicación sin el apoyo de su propia herramienta; el proceso para traducir requisitos al modelo causa-efecto no está documentado; no tiene incluida ninguna plantilla, ni propuesta formal de alguna para redactar requisitos; la actividad en la cual se hace la revisión de las ambigüedades se describe muy pobremente y, debido a que se realiza por personas, tiende a no ser verdaderamente objetiva en su resultado; la ausencia de métodos formales imposibilita su automatización; no se encuentra referencias a casos de aplicación ni de éxito; los ejemplos confunden en vez de clarificar; se duplica un proceso ya que el diagrama causa-efecto y la tablas contienen la misma información; la documentación de los procesos de aplicación es pobre.

Ofrece las ventajas de contar con una herramienta de soporte y de detallar cómo se puede integrar las actividades de generación de casos de prueba en un proceso de desarrollo.

2.5 Use case derived test cases [21]

Esta propuesta parte de un caso de uso descrito en lenguaje natural e indica toda la información que debe contener: nombre, descripción, requisitos, pre y pos-condiciones y flujo de eventos, y entrega un conjunto de casos de prueba, también descritos en lenguaje natural, que define las acciones y verificaciones que realiza en el sistema.

El procedimiento consiste en identificar los caminos de ejecución, que son todos los caminos posibles, a partir del flujo de eventos de cada caso de uso; posteriormente, cada camino identificado se transforma en un caso de prueba descrito en lenguaje natural.

No se encuentra una ventaja significativa respecto de las demás propuestas; no tiene suficiente documentación; ofrece escalabilidad nula; no describe claramente cómo diferenciar los caminos; no define con claridad el manejo de los requisitos; no es posible seleccionar los casos de prueba (no es aplicable a grandes sistemas); no tiene una referenciación suficientemente amplia que la soporte; y la descripción en lenguaje natural de los casos de uso imposibilita su automatización.

2.6 Testing from use cases using path analysis technique [22]

Se inicia con un caso de uso descrito en lenguaje natural, desde el cual se elabora un diagrama de flujo con los posibles caminos que deben ser probados para recorrerlo. Esos caminos deben ser analizados para luego asignarles una puntuación de acuerdo a su importancia y frecuencia; están descritos en lenguaje natural sin adoptar una presentación formalizada; los caminos mejor evaluados se convierten en los casos de prueba a aplicar, representados en un diagrama de flujo. Puede existir más de un caso de prueba probando un mismo camino, ya que es necesario añadirle valores de prueba a cada uno.

El proceso que describe es sencillo y relativamente fácil de implementar; la forma de cómo descartar caminos, que no aportan a la prueba, está bien descrita; es posible probar el comportamiento de cada caso de uso seleccionado de forma muy completa, e incluye un ejemplo paso a paso de aplicación.

No indica cómo describir los casos de uso, ni utiliza reglas para hacerlo, pero sí indica cuál es la información que debe incluirse al redactarlos; además, el análisis de los atributos no está demarcado y es posible hacerlo con cuántos se desee, tampoco indica cómo realizar la puntuación de los mismos; no detalla cómo aplicar los casos de prueba, ni cuál será el posible resultado que se alcance, o su estructura. No referencia proyectos reales en los que se haya aplicado; el uso de un lenguaje natural para describir casi todo el proceso da pie para ambigüedades y se convierte en un problema al momento de automatizar las pruebas; si un caso de uso es dependiente de otro, no está documentado como probarlo; aunque es una propuesta sencilla, es difícil estructurarla para aplicar en sistemas más grandes.

2.7 Requirements by contract [23]

La propuesta parte de un diagrama de casos de uso en notación UML y propone cuatro criterios por medio de los cuales es posible recorrer el modelo para obtener los casos de prueba. Se encuentra dividida en dos momentos: en el primero se hace una extensión de los casos de uso UML mediante un lenguaje de contratos que incluye pre y pos-condiciones, expresadas mediante proposiciones, y sus parámetros, que en conjunto permiten expresar las dependencias existentes entre los casos de uso; en el segundo momento se detalla la generación automática de los casos de prueba desde la extensión de los casos de uso.

Como resultado se obtiene un modelo de casos de uso extendido con contratos, expresados como caminos que recorren la ejecución de las secuencias de casos de uso que satisfacen las pre y pos condiciones, y un conjunto de casos de prueba para verificar la implementación del modelo que genera. Este modelo se expresa en un diagrama que refleja el comportamiento del sistema según los casos de uso diseñados. El estado del sistema se representa por cada nodo del modelo, determinado por las proposiciones, y las instancias se representan por cada transición.

No detalla cómo generar los casos de prueba desde las instancias por medio de las herramientas de generación de pruebas; no es posible desarrollar pruebas que verifiquen aisladamente el comportamiento de cada caso de uso; los casos de uso se extienden sin respetar el estándar UML; no hay forma de saber el número de parámetros que debe utilizar cada caso de uso, y no se encuentra una referencia de cómo implementar las pruebas.

Su fortaleza se refleja en la diversidad de criterios de cobertura y su herramienta experimental de soporte (de libre descarga); las pruebas se pueden generar desde la secuencia de casos de uso; los contratos son muy flexibles para expresar dependencias entre los casos de uso. El mismo equipo de autores publicó un trabajo en el que describen cómo aplicarla en una familia de productos software [24].

2.8 Category partition method [25, 26]

Describe cómo generar los casos de prueba a partir de los casos de uso de una familia de productos [27], y extiende la notación de esos casos de uso mediante una plantilla en lenguaje natural [5], que contiene las características comunes a los productos de la familia, lo mismo que los puntos en que cada producto varía de los demás. Este proceso es una adaptación del

propuesto por Ostrand y Balcer [28], actualizado para trabajar con especificaciones de familia de productos que se pueden modelar mediante casos de uso.

El proceso comienza con la representación de los casos de uso del sistema; se determina los requisitos funcionales y los rangos de datos que cada uno podría tomar, una tarea que realizan los encargados de las pruebas con base en el conocimiento del sistema y su experiencia; se determina las restricciones en cada uno de los rangos que genera errores y que reduzca el número de casos de prueba; se redacta las especificaciones de la prueba, que es un documento en el que se describe la plantilla con la información recolectada en los pasos anteriores; se genera el contexto de la prueba, que consiste en una combinación de los valores encontrados en los rangos de datos; se traduce esa combinación de valores a un lenguaje ejecutable y se reúne aleatoriamente para ejecutarse en el sistema.

El resultado final de la propuesta es un conjunto de casos de prueba específico para cada producto y el conjunto de casos de prueba común para los productos de la familia. Estos casos de prueba se describen de forma abstracta, por lo que deben refinarse para obtener casos de prueba posibles de ejecutar en el sistema. Puede adaptarse para aplicar en sistemas que no pertenecen a una familia específica; aunque es una propuesta para generar conjuntos de valores para las pruebas, muchos de los pasos presentan ambigüedad en su definición, ya que quedan supeditados a la experiencia y conocimiento del sistema por parte del equipo de pruebas, por lo que la información acerca de la cobertura de las pruebas tampoco se ofrece adecuadamente; no es posible su automatización total, y no referencia una herramienta que la soporte.

La propuesta describe un ejemplo práctico de aplicación, es posible verificar el comportamiento de los casos de uso, así como su dependencia con otros; se puede aplicar desde comienzos del proceso del proyecto, y también reduce el número de casos de prueba generados a través de las restricciones que tiene en cuenta.

2.9 Requirements to testing in a natural way [29]

Según sus autores es un analizador de requisitos que puede ser utilizado para generar pruebas de caja blanca y caja negra; está conformada por seis actividades dividas en dos bloques: el primero lo conforman cinco actividades: 1) se redacta los requisitos en lenguaje natural y en párrafos estructurados; 2) se hace una identificación de cada frase en los párrafos de descripción de los casos de uso; 3) desde cada una de las frases se genera un árbol sintáctico con sus respectivas anotaciones; 4) a partir de cada árbol se obtiene la estructura de representación del discurso [30], en la que se identifica la semántica de cada elemento del árbol; y 5) se refina esta estructura y se elimina las ambigüedades existentes, luego se traduce automáticamente al lenguaje MONA [31], con lo que se genera una máquina de estados finitos.

En el segundo bloque se recorre la máquina de estados finitos para generar los casos de prueba. Se ofrece la posibilidad de recorrer la máquina de distintas formas, por lo que es posible obtener varios conjuntos de casos de prueba que generan explosión de los mismos. Para evitarlo propone preguntar al usuario por los requisitos críticos, y luego reducir los recorridos a ellos.

Aunque es una propuesta bien documentada, no incluye ejemplos de las pruebas generadas ni cómo se recorren y obtienen las pruebas a partir de la máquina de estados; al construir el árbol sintáctico no es posible clarificar entre verbos, sujetos y otros componentes de las frases que describen los casos de uso; no explica si es necesario recorrer mediante pruebas todas las frases de la descripción del caso de uso.

Su principal ventaja es ser de las pocas que detalla un método para procesar requisitos descritos en lenguaje natural, lo que permite automatizar la generación de las pruebas; aunque no se encontró un caso práctico de aplicación, existe herramientas libres que permiten automatizar parte de las actividades y, para las otras, existe herramientas desarrolladas por los mismos autores, a las que no es posible acceder, por lo que no fue posible verificar su aplicación práctica.

2.10 A model-based approach to improve system testing of interactive applications [32]

Esta propuesta tiene su origen en las investigaciones de la empresa Siemens, las cuales recopila y amplía Ruder [33]. Parte de la documentación en lenguaje formal de los casos de uso y, como resultado de su aplicación, se obtiene el conjunto de pruebas para la interfaz gráfica del sistema.

Los pasos son los siguientes: 1) se modela el comportamiento del sistema mediante un diagrama de actividades UML que describe el comportamiento interno de los casos de uso, y especifica los requisitos de prueba (conjunto de estereotipos que permiten interpretar cada actividad); estos estereotipos indican la pertenencia de las actividades del usuario, del sistema u otro diagrama de actividades, y se interpretan para construir pruebas ejecutables por el generador de pruebas; 2) se diseña los casos de prueba mediante *scripts* de prueba, que son la descripción de los diagramas de actividades en lenguaje Test Specification Language TSL [34]; luego, mediante Test Development Environment TDE [33] se traduce de TSL a guiones de prueba, con lo que obtiene un conjunto de *scripts* que puede ejecutarse en una herramienta de verificación de interfaces gráficas; y 3) luego de construir el conjunto de guiones, es posible ejecutarlos sobre el sistema que se está probando, proceso que refina y mejora los mismos *scripts* a medida que se ejecutan.

Esta propuesta es de las pocas que describe cómo generar pruebas ejecutables; utiliza lenguaje formal que, unido al uso de *scripts* mediante estereotipos escalables, facilita su automatización. Sus inconvenientes se reflejan en que se centra en la interfaz gráfica, imposibilitando su aplicación en otras interfaces; el proceso de cómo se obtiene los diagramas de actividades desde los casos de uso no se detalla lo suficiente, lo mismo que cómo se asigna los estereotipos a cada actividad; no especifica si el proceso de pasar los diagrama de actividades a TSL debe hacerse manual o si es automatizable; no es claro si se requiere un solo diagrama de actividades para todo el proceso o es necesario diseñar uno por cada caso de uso; los diagramas de actividades de cada caso de uso aparecen independientes y no se detalla qué hacer con las relaciones entre ellos; no se encuentra una herramienta que permita aplicar la propuesta en un ambiente de laboratorio o de evaluación.

3. CONCLUSIONES

Probar un sistema desde la óptica de las pruebas funcionales es verificar que los requisitos de la especificación funcional se han implementado bien, por lo que deben constituirse en la base sobre la que se diseñan los casos de prueba del sistema.

Diseñar el sistema teniendo como base su especificación funcional es una tarea en la que se construye un modelo formal (o semi-formal) que, de acuerdo con los datos en los requisitos, debe expresar lo que se espera del comportamiento del sistema. Dado que la base sobre la que se diseña son los requisitos en lenguaje natural, el proceso no puede realizarse automáticamente, pero sí sistemáticamente, mediante el seguimiento de pasos y fases definidos con claridad.

Luego del diseño, en la aplicación de la cobertura de la prueba es necesario seleccionar un criterio mediante el cual se pueda generar los casos de prueba, que posteriormente se aplicarán al diseño

del modelo. En este análisis el criterio que tiene más acogida en las propuestas es el de todos los posibles caminos de ejecución; aunque existe otros criterios que también son viables y que, al igual que los analizados, pueden realizarse automáticamente con una herramienta software.

Otra característica de las propuestas analizadas es que los valores de los datos de entrada al sistema hacen parte del proceso de prueba, por lo que cualquier propuesta debe tener en cuenta cómo generarlos. Para este caso ese proceso debe describirse más detalladamente, ya que la descripción es muy pobre en la mayoría de las propuestas.

Los resultados esperados son un elemento imprescindible en la descripción de las propuestas de este análisis, y consiste en poder determinar cuál será la respuesta del sistema para la ejecución del conjunto de casos de prueba seleccionado; estos resultados se obtienen luego de aplicar automáticamente las pruebas a una simulación del modelo del sistema. En este análisis, un porcentaje muy pequeño se ocupa de este tema y parece no serles de utilidad para diseñar el conjunto de casos de prueba.

El paso final en los procesos de prueba analizados es el de ejecutar los casos de prueba sobre el sistema objeto, pero ninguna de las propuestas ofrece el detalle de cómo generarlo mediante un formalismo que pueda traducirse fácilmente a código, por el contrario, la mayoría describen cómo traducirlos a lenguaje natural.

El análisis efectuado a las propuestas para diseñar casos de prueba a partir de la especificación funcional, sustenta la falta de un trabajo más profundo en procura de hallar una propuesta con más integración, y que debe desarrollarse a partir de lo siguiente:

- Los puntos comunes de las propuestas que se analizan en este documento, como son: obtener un conjunto de casos de prueba que de alguna manera garantice que el sistema cumple con las especificaciones funcionales; partir de los requisitos funcionales para generar los casos de prueba; utilizar el análisis de caminos o estados posibles; tener en cuenta que los requisitos funcionales necesariamente no cumplen requisitos formales; generar el conjunto de casos de prueba de manera automática y sistemática a partir de los requisitos funcionales, mediante alguna herramienta software, y validar los requisitos funcionales desde las primeras fases del desarrollo. Estos puntos deben ser la base desde la que se puedan corregir las falencias encontradas y para potenciar sus fortalezas en el nuevo proyecto.

- En lo que respecta al modelo de comportamiento del sistema se debe identificar claramente cuál es la información que, contenida en la especificación funcional, permita generarlo y luego obtener los casos de prueba.

- Debe diseñarse y describirse una plantilla en la que se estandarice la descripción de los requisitos funcionales en lenguaje natural, para lo que se puede pensar en utilizar una pre-existente, como la que describe [5], o desarrollar otra.
- Dado que los casos de uso no son fijos, debe detallarse cuál es el grado de refinamiento necesario para generar los casos de prueba; mirar por ejemplo el refinamiento propuesto por Dustin et al. [35].

- Para obtener una propuesta más robusta es necesario incluir, además de los funcionales, otro tipo de requisitos, como los de almacenamiento; ya que los valores de salida son tan importantes en la realización de la prueba, con estos requisitos es posible darles mayor cobertura y eficacia a los casos de prueba.

- En las propuestas analizadas se trabaja con un único modelo del sistema en la generación de los casos de uso, lo que crea dificultades para analizar las interacciones entre éstos; poder contar con modelos alternos, o sub-modelos basados en un diagrama de estados, facilita el análisis del comportamiento del sistema desde la especificación funcional.

- Otro asunto importante es el relacionado con el criterio de cobertura, en especial el de cómo utilizar la prioridad de los casos de uso al momento de generar interacciones o secuencias; puede tenerse en cuenta algunas propuestas desde la Ingeniería de Requisitos, como las de Robinson [36], Riebisch et al. [37] y Escalona [38].

- En lo que respecta a los valores de prueba, debe diseñarse un conjunto de reglas concreto y sistemático que permita reducir el número de decisiones que toman los probadores, a lo cual ayudará lo ya expuesto de incluir el proceso los requisitos de almacenamiento.

- El número de pruebas a ejecutar por cada escenario posible de aplicación es una cuestión fundamental, cuando se hace análisis de caminos, y para solucionarlo es conveniente pensar en un conjunto concreto de valores y generar un escenario de prueba por cada combinación posible.

- Es necesario recurrir a los lenguajes formales para expresar los escenarios de prueba, ya que con ellos es posible generar automáticamente los casos de prueba; para esta tarea es conveniente revisar trabajos como los de Clarke y Wing [39], Lamsweerde [40], Juristo et al. [41], Clermont y Parnas [42], Beckert et al. [43], Dasso y Funes [44] y Kaner [45]. Esta forma de trabajo permitirá cuantificar el grado de cobertura de la prueba de forma automatizada, lo que aleja la toma de decisiones del actor humano.

- Validar el conjunto de casos de prueba generado debe ser una meta en la nueva propuesta, por ejemplo, mediante la cobertura de los requisitos, si la cobertura es la deseada el conjunto es válido.

- Es necesario contemplar la posibilidad de automatizar cada una de las actividades en el proceso de generación del conjunto de casos de prueba; esta meta debe considerarse prioritariamente ya que es una falencia que resta credibilidad y eficacia a las analizadas. Igualmente, es importante considerar la posibilidad de generar una herramienta de soporte a la propuesta, de tal manera que sea posible aplicarla a trabajos y en situaciones reales.

REFERENCIAS

[1] Beizer B. (1990). Software testing techniques. International Thomson Computer Press.
[2] Myers G. (1979). The art of software testing. Wiley.
[3] Myers G. (2003). Principles of functional verification. Newnes.
[4] Fröhlich P. y Link J. (2000). Automated test case generation from dynamic models. Lecture Notes in Computer Science 1850, 472-491.
[5] Cockburn A. (2000). Writing Effective use cases. Addison-Wesley.
[6] Fröhlich P. y Link P. (1999). Modeling Dynamic Behaviour Based on Use Cases. En 3rd International Software Quality Week Europe QWE. Brussels, Belgium.
[7] Fikes R. y Nilsson N. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence 2(3-4), 189-208.
[8] Hugger J. (2001). Wellposedness of the boundary value formulation of a fixed strike Asian option. Journal of Computational and Applied Mathematics 185(2), 460-481.

[9] Copeland L. (2004). A practitioner's guide to software test design. Artech House.
[10] Myers G. (2004). The art of software testing. John Wiley.
[11] Schroeder P. y Korel B. (2000). Black-box test reduction using input-output analysis. En International Symposium on Software Testing and Analysis. New York, USA.
[12] McGee P. y Kaner C. (2004). Experiments with high volume test automation. SIGSOFT Software Engineering Notes 29(5), 1-3.
[13] Hierons R. (2006). Avoiding coincidental correctness in boundary value analysis. ACM Transactions on Software Engineering and Methodology 15(3), 227-241.
[14] Krishnan R. et al. (2007). Combinatorial testing: learnings from our experience. SIGSOFT Software Engineering Notes 32(3), 1-8.
[15] Tuya J. et al. (2008). A controlled experiment on white-box database testing. SIGSOFT Software Engineering Notes 33(1), 1-6.
[16] Masri W. et al. (2009). An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. En International Workshop on Defects in Large Software Systems. New York, USA.
[17] Heumann J. (2002). Generating test cases from use cases. Journal of Software Testing Professionals the Rational Edge (September), 3-14.
[18] Mogyorodi G. (2001). Requirements-based testing: An overview. En 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. Santa Barbara, USA.
[19] Mogyorodi G. (2002). Requirements-based testing: Ambiguity reviews. Journal of Software Testing Professionals (December), 21-24.
[20] Mogyorodi G. (2003). What is requirements-based testing? Crosstalk: The Journal of Defense Software Engineering 16(3), 12-15.
[21] Wood D. y Reis J. (2002). Use case derived test cases. Software Quality Engineering for Software Testing Analysis and Review (Memories), 235-244.
[22] Naresh A. (2002). Testing from use cases using path analysis technique. En International Conference On Software Testing Analysis & Review. Washington, USA.
[23] Nebut C. et al. (2003). Requirements by contract allow automated system testing. En 14th International symposium of Software Reliability Engineering. Denver, USA.
[24] Nebut C. et al. (2004). A requirement-based approach to test product families. Frank van der Linden 30(14), 198-210.
[25] Bertolino A. y Gnesi S. (2003). Use Case-based testing of product lines. ACM SIGSOFT Software Engineering Notes 28(5), 355-358.
[26] Bertolino A. y Gnesi S. (2004). PLUTO: A test methodology for product families. Lecture Notes in Computer Science 30(14), 181-197.
[27] Bertolino A. et al. (2002). Use case description of requirements for product lines. En REPL'02. Essen, Germany.
[28] Ostrand T. y Balcer M. (1988). The category-partition method for specifying and generating functional tests. Comunications of the ACM 31(6), 676-686.
[29] Boddu R. et al. (2004). RETNA: from requirements to testing in a natural way. En 12th IEEE International Requirements Engineering Conference. Kyoto, Japan.
[30] Blackburn P. y Bos J. (1994). Working with discourse representation theory: An advance course in computational semantics. Stanford CSLI Publications.
[31] Henriksen J. et al. (1995). MONA: Monadic second-order login in practice. En Tools and Algorithms for the Construction and Analysis of Systems Conference. Warsaw, Poland.
[32] Hartmann J. et al. (2004). UML-based test generation and execution. En Workshop on Software Test, Analyses and Verification. Banff, Canada.
[33] Ruder A. (2004). UML-based test generation and execution. Siemens Corporate Research.
[34] Balcer M. et al. (1990). Automatic generation of test scripts from formal test specifications. ACM SIGSOFT Software Engineering Notes 14(8), 210-218.
[35] Dustin E. et al. (2002). Quality Web systems. Addison-Wesley.
[36] Robinson H. (2000). Intelligent test automation. Software Testing & Quality Engineering 2(5), 24-32.
[37] Riebisch M. et al. (2003). UML based statistical test case generation. Lecture Notes in Computer Science 2591, 394-411.

[38] Escalona M. (2004). Modelos y técnicas para la especificación y el análisis de la navegación en sistemas software. Disertación doctoral. Universidad de Sevilla.
[39] Clarke E. y Wing J. (1996). Formal methods: State of the art and future directions. ACM Computing Surveys 28(4), 626–643.
[40] van Lamsweerde A. (2000). Formal specification: A roadmap. En Conference on The Future of Software Engineering. Limerick, Ireland.
[41] Juriso N. et al. (2004). Reviewing 25 years of testing technique experiments. Empirical Software Engineering 9(1-2), 7-44.
[42] Clermont M. y Parnas D. (2005). Using information about functions in selecting test cases. En 1st International Workshop on Advances in Model-Based Testing. St. Louis, USA.
[43] Beckert B. et al. (2006). Intelligent systems and formal methods in software engineering. IEEE Intelligent Systems 21(6), 71-81.
[44] Dasso A. y Funes A. (2007). Verification, validation and testing in software engineering. Idea Group Publishing.
[45] Kaner C. (2003). What Is a Good Test Case? En STAR East 2003. Orlando, USA.

CAPÍTULO XVII
Una revisión a la realidad de la automatización de las pruebas del software[1]

Edgar Serna M.[1]
Raquel Martínez M.[2]
Paula Tamayo O.[2]
[1]Instituto Antioqueño de Investigación
[2]Institución Universitaria de Envigado

Probar el software es una de las actividades más importantes en el ciclo de vida del desarrollo, pero tradicionalmente se ha llevado a cabo al final del proceso, cuando el producto está terminado y está a punto de ser liberado. La complejidad del software actual exige que la prueba se ejecute de forma paralela al desarrollo, de tal manera que los errores se encuentren a tiempo y se puedan corregir a bajo costo. La automatización de las pruebas surgió como una alternativa para agilizar su ejecución, a la vez que para mejorar la fiabilidad del producto y su calidad. Pero, aunque su estudio e investigación se inició hace más de medio siglo, su progreso todavía no satisface la demanda por el mejoramiento de la calidad. En este capítulo se presenta los resultados de una revisión integradora de la literatura, que se realizó con el objetivo de establecer una visión general de las experiencias reportadas sobre la automatización de las pruebas. Se encontró que todavía no hay acuerdo acerca de su definición, y que las ventajas y limitaciones son prácticamente opiniones. Además, que muchas empresas asumen la automatización como un remplazo total de las pruebas manuales, aunque su nivel de madurez todavía no lo permita.

[1] Publicado en la Revista Computación y Sistemas 23(1), 169–183. 2019.

INTRODUCCIÓN

Aunque en la literatura se discute acerca de la automatización de las pruebas su definición todavía se concentra en la creación de un mecanismo para automatizar casos de prueba, lo que significa que se utiliza un software especial para ejecutarlos a través de secuencias de comandos preparados, en lugar de un probador. En términos generales se asume que con la automatización las actividades de pruebas manuales tienden a ser sustituidas por actividades automatizadas [1], lo que significa que no se requiere la participación humana para generar una prueba [2].

A diferencia de las pruebas manuales, en las que en todo el proceso se requiere que un probador interactúe con cada caso de prueba para analizarlo y reportar los resultados, la prueba automatizada consiste en utilizar un software especial para controlar la ejecución de las pruebas y la comparación de los resultados reales con los resultados esperados [3]. Hoffman [4] declara que las pruebas de software se consideran automatizadas cuando un mecanismo libre prueba el funcionamiento de los casos de prueba. En [5] también se describe la automatización como la tarea de crear una representación mecánicamente interpretable de un caso de prueba manual. Para Sommerville [6] son casos de prueba que se ejecutan automáticamente para acelerar el proceso de prueba, y una de las razones por las que son importantes es que garantizan la fiabilidad del software, especialmente con las actualizaciones.

Diversos autores e investigadores recalcan la importancia de las pruebas del software. Una de las razones es que a través de ellas es posible detectar los defectos y reducir los riesgos asociados [7]. Además, el aseguramiento de la calidad es un factor crítico para el éxito del desarrollo de sistemas, es la clave para la satisfacción del cliente y tiene un impacto directo en el costo y el desarrollo del producto [8].

Cuando las pruebas se aplican adecuadamente pueden garantizar que el software y los procesos en el ciclo de vida se ajustan a las necesidades específicas [9]. Pero tradicionalmente se conciben como una entidad separada que funciona independientemente del área de desarrollo, aunque las personas que las llevan a cabo son responsables de la definición del proceso y del seguimiento de los detalles de su ejecución [9].

Debido a su importancia y necesidad debe ser un procedimiento involucrado en todas las fases del ciclo de vida [10-12]. Por su parte, el proceso de automatización es crucial, y aunque las pruebas parezcan rezagadas con respecto al ritmo de escritura del código y el número de líneas que cada programador desarrolla por día, la impresión es que permanece relativamente fijo [13], mientras los reprocesos se incrementan debido a la complejidad de los sistemas actuales.

El término pruebas manuales, al igual que automatizadas, significa algo más que la ejecución de casos de prueba, y la idea de que este proceso pueda hacerse sin la intervención humana es una cuestión de amplio debate en la comunidad [33]. Si bien es cierto que los sistemas complejos requieren más de la automatización, mucho software trivial y con interfaces sencillas se puede probar sin estas herramientas, porque el costo es un ítem que se debe tener en cuenta. Por eso es que en todo proyecto de desarrollo se debe tener la posibilidad de elegir entre unas u otras para garantizar la fiabilidad del producto.

La automatización acelera el proceso de prueba, y una de las razones por las que es importante es que asegura de mejor forma la fiabilidad del software, especialmente cuando se realiza actualizaciones [6]. Otra manera de lograrlo es utilizando pruebas redundantes, lo que disminuye los costos de mantenimiento y asegura la integridad del conjunto de pruebas [14].

Se ha demostrado que una vez que se escribe los casos de prueba es necesario actualizarlos cada vez que se realiza cambios en el software. Si este proceso no se ejecuta con cuidado, la integridad de las suites de prueba puede ser cuestionable. La investigación en este campo es activa y los resultados se difunden en trabajos de todo tipo, en los que se compara técnicas, herramientas y prácticas de prueba. Para Bertolino [15] la prueba se ha convertido en una actividad esencial en la Ingeniería del Software, y propone una hoja de ruta que puede ser establecida a través de un conjunto de cuatro objetivos finales y alcanzables. Uno de ellos es la cantidad de pruebas que se debe o puede hacer, donde el factor humano es un recurso fundamental, por lo que la capacidad, el compromiso y la motivación de los probadores pueden hacer gran diferencia entre el éxito o el fracaso del proceso. Las personas ejecutan pruebas manuales sin necesidad de utilizar ninguna herramienta automatizada o cualquier secuencia de comandos. El probador asume el papel de un usuario final y prueba el software para identificar comportamientos inesperados del programa.

En este trabajo se presenta los resultados de una investigación cuyo objetivo fue establecer el dinamismo con que se divulga acerca de la automatización de las pruebas. En primer lugar, la idea fue reforzar los conocimientos y la experiencia existentes mediante la identificación de los temas y conceptos investigados; en segundo lugar, y con base en esos temas y conceptos, establecer una visión general de la experiencia existente sobre la automatización. Para lograrlo se identificaron y revisaron los reportes de experiencias y las publicaciones relacionadas.

1. TRABAJOS RELACIONADOS

Dustin et al. [16] afirman que el desarrollo de la automatización de las pruebas también se debe a la aplicación de desarrollo rápido de aplicaciones RAD, y a su popularidad. Para cumplir la mayor cantidad de actividades relacionadas con las pruebas se debe utilizar herramientas automatizadas, porque a menudo las manuales son laboriosas y propensas a errores, y simplemente no pueden competir con la calidad de la prueba automatizada, especialmente cuando hay que tener en cuenta el cronograma del proyecto. Estos autores no tienen en cuenta el costo de la automatización, debido a que su trabajo se centra en productos generalmente pequeños, pero otro punto de vista sería analizarla para productos complejos y críticos, donde la calidad y fiabilidad es el objetivo central.

El tema del trabajo de Bertolino [15] es la automatización de las pruebas del software, vista como un enfoque de validación extendido en la industria, pero que sigue siendo en gran medida *ad hoc*, costoso y con una eficacia impredecible. De hecho, automatización es un término amplio que abarca una variedad de actividades a lo largo del ciclo de vida y más allá, y que tiene diferentes objetivos. Por lo tanto, cualquier investigación relacionada con la automatización de las pruebas se enfrenta a una serie de desafíos. La autora propone un plan de trabajo coherente para afrontar los más relevantes, y su punto de partida lo constituye algunos logros importantes del pasado, sustentados en los cuatro objetivos que identifica, pero que siguen siendo sueños inalcanzables.

De acuerdo con Bavin [17], para automatizar las pruebas existen herramientas eficientes y efectivas, pero debido a que su valor es alto, tienen que ayudar a realizar actividades de prueba más rápidas y eficaces que las manuales. Entre sus beneficios se encuentra que la información sobre los errores encontrados se puede reutilizar y que la búsqueda puede comenzar mucho antes en el ciclo de vida. Esto es importante, porque en el desarrollo de software hay que tener en cuenta el tiempo y los aspectos de presupuesto, y el uso de estas herramientas ayuda a acelerar el proceso. El autor no hace una comparación a la efectividad de esas herramientas y se limita a describir la valoración subjetiva que algunos autores publican.

Whyte y Mulder [18] proponen que las pruebas de software son uno de los principales métodos utilizados en la validación y verificación de la producción y desarrollo de software en la industria. La mayoría las ve como un método clave para la consecución de la calidad y la fiabilidad del software y la satisfacción del cliente. Sin embargo, es un proceso costoso que representa alrededor del 50% del valor de desarrollo de sistemas. Debido a que en los últimos años la Ingeniería del Software ha estado bajo presión para cumplir con el tiempo, el costo y las limitaciones del producto, es fundamental identificar y aplicar herramientas que reduzcan el impacto negativo de la prueba, y que incrementen su eficacia. Las principales conclusiones de este estudio es que no hay un enfoque individual que produzca resultados satisfactorios, pero una combinación de dos o más tipos de pruebas automatizadas podría incrementar la eficacia y mitigar los efectos de sus limitaciones.

Para Rafi y sus compañeros [19] existe una brecha documentada entre los puntos de vista académicos y profesionales acerca de las pruebas del software. En su trabajo tratan de cerrarla mediante una investigación a ambos puntos de vista con respecto a los beneficios y las limitaciones de la automatización. Estudian la visión académica mediante una revisión sistemática de la literatura, y la profesional la evalúan con una encuesta. En la academia la evidencia publicada es poco profunda y los beneficios se sustentan a partir de fuentes como experimentos y estudios de casos, mientras que las limitaciones lo hacen a partir de reportes de experiencias. Los beneficios de la automatización para los profesionales están relacionados con la reusabilidad, la repetitividad, la cobertura y el esfuerzo ahorrado. Las limitaciones son la alta inversión inicial en su configuración, la selección de las herramientas y la capacitación del personal.

Majikes et al. [20] opinan que los dispositivos médicos controlados por software evolucionan desde monolíticos a sistemas médicos modulares cyber-físicos, y que en ellos el software es un componente crítico, y que al añadir mayor complejidad se incrementa la probabilidad de fallos en los mismos. En este caso, las pruebas automatizadas se han utilizado como un medio para asegurar la calidad del software, lo que es conveniente para este tipo de sistemas. En su trabajo, presentan una revisión a las técnicas de pruebas automatizadas existentes que exponen fallos en estos dispositivos. En particular, categorizan las fallas y luego aplican el método de revisión sistemática de la literatura para comparar los estudios existentes contra su categorización. Los autores sugieren mejoras que pueden ayudar a futuros estudios de investigación.

Para comprender mejor los retos que enfrentan los usuarios y los desarrolladores de pruebas automatizadas, Wiklund et al. [21] realizaron una investigación empírica sobre un panel de discusión que soporta un *framework* de automatización de pruebas, y que tiene cientos de usuarios. Encontraron que gran parte de los problemas y solicitudes de ayuda se relacionan con ambientes TI centralizados, y con el comportamiento erróneo en el uso del *framework* y sus componentes. En cuanto a las solicitudes de ayuda, la mayoría se refiere al diseño de *scripts* de prueba, y no a las áreas que parecen ser más problemáticas. A partir de estos resultados observaron una clara necesidad de simplificar el uso, la instalación y la configuración de la prueba automatizada. Encontraron que los problemas detectados sugieren que los probadores necesitan desarrollar habilidades diferentes, lo que históricamente se ha asumido como cubierto. Por último, proponen nuevas investigaciones acerca de los beneficios de la automatización y de su implementación y uso eficiente en entornos TI.

El propósito del estudio de caso presentado por Guan [22] es entender el punto de vista de las organizaciones que utilizan pruebas automatizadas, y cómo lo relacionan en la literatura. Este autor describe y analiza los factores clave para tener en cuenta al configurar una prueba automatizada, además de los riesgos que implica. Encontró que la cognición de estas pruebas no

se limita solamente a conocer su definición y los beneficios que pueden ser introducidos en la organización, sino que también tienen que centrarse en el ámbito de su aplicación y las condiciones previas. Determinó que es necesario gestionar consideraciones clave, tales como la resistencia de las personas, el proceso de trabajo y entrenamiento, y algunas otras.

2. MÉTODO

Una revisión de la literatura se realiza con el fin de establecer una comprensión del estado de la investigación sobre un área específica. El objetivo es identificar los temas que se han abordado y sistematizar los resultados relacionados. Existe diversas propuestas de clasificación para estas revisiones, pero el tipo que se aplicó en esta investigación es una revisión integradora [23]. De acuerdo con esta propuesta, el proceso consiste en identificar las publicaciones de relevancia, para evaluarlas y seleccionarlas para su revisión y sistematización, de tal manera que se pueda identificar y estructurar los conceptos centrales comunes.

Por lo tanto, una revisión integradora desarrolla una visión general de la investigación existente, que inicialmente no está relacionada y que servirá de base para futuras investigaciones. Desde este punto de vista se realiza tres actividades: 1) identificar la literatura potencialmente relevante, 2) evaluar la pertinencia y calidad de la literatura seleccionada para el análisis y la síntesis, y 3) analizar y sintetizar la literatura seleccionada.

1. Para la actividad 1 se buscaron los reportes de investigación en las bases de datos en línea. Esto incluyó búsquedas en ScienceDirect, ACM Digital Library, IEEE Explore y Web of Science. En conjunto estos sitios cubren un amplio rango de revistas científicas, libros y conferencias relacionadas con el área de interés. También se realizaron búsquedas directas en el motor de búsqueda, porque algunas fuentes pueden no estar indexadas. En todos los casos se realizaron búsquedas utilizando los siguientes términos clave: *Automated software testing, automated testing, test automation, test automation software* y *software testing automation*. Esta actividad incluyó una búsqueda en la línea de tiempo desde finales de los años 90, dado que una publicación requiere entre tres y cuatro años para que la comunidad la analice y referencie. Se revisaron el título y la fuente (revista, libro o conferencia) con el objetivo de excluir los resultados no-relevantes para el área de interés. No se tuvo en cuenta las editoriales, prólogos, resúmenes de artículos, entrevistas, noticias, críticas u opiniones, correspondencia, discusiones, comentarios, cartas de los lectores, resúmenes de talleres, paneles ni sesiones de posters. El proceso dio como resultado 112 publicaciones.

2. En la actividad 2 se evaluó cada trabajo para asegurar que sí explica o discute el concepto de o el uso de la automatización de las pruebas del software, y que reporta hallazgos empíricos. Con base en una revisión de los resúmenes se eliminaron 42 trabajos, los demás constituyeron la base del estado de la práctica de esta investigación. Al estudiar el texto completo se eliminaron 17 más, considerados no relevante para la revisión, dejando 53 trabajos en total para la síntesis.

3. En la actividad 3 se hizo la síntesis de los trabajos. La colección resultante difiere con respecto a la variación temática, el tipo de estudio y la calidad científica. Se encontró que 18 reportan resultados de estudios empíricos, en el sentido que describen el método de estudio, el análisis y la validez de los resultados. El resto de los documentos puede ser clasificado como informes de experiencias o elaboraciones de conceptos o ideas. Para asegurar que no se pasaran por alto las conclusiones y las tendencias potencialmente relevantes en este campo de investigación, se decidió mantener también los trabajos estrictamente no-empíricos, tales como informes de experiencias ligeras.

De esta forma se presenta una visión general del estado de la investigación en la automatización de las pruebas, en la que se muestra los temas y conceptos que se han abordado, y un resumen de esa experiencia. Aunque en principio es conocimiento útil, es necesario verificarlo y ampliarlo con estudios empíricos. También se identificaron temas y conceptos que poco se han investigado hasta ahora, pero que deben ser incluidos en futuras investigaciones. Esta consideración se basa en parte en lo que describen los trabajos y en las conversaciones con profesionales en el área.

El análisis de la literatura seleccionada se realizó mediante comparación constante, donde los conceptos y temas centrales emergen a través de análisis repetidos de la literatura revisada. Parte de la literatura incluida en la revisión no se pudo clasificar como estudios empíricos rigurosos, sin embargo, dado que la automatización de las pruebas es una práctica en ascenso, también se incluyeron los trabajos no-empíricos, en los que se valoró la importancia de lo que reportan acerca de la situación actual del tema.

3. RESULTADOS

Como la automatización de las pruebas del software gana popularidad, el cuerpo de la literatura sobre el tema ha ido creciendo de forma constante en los últimos años. Los resultados de esta investigación se estructuran de acuerdo con los tópicos identificados más destacados y relevantes. Esto no quiere decir que la lista de temas que se presenta aquí sea el conjunto final de lo más importante en automatización de las pruebas del software, simplemente refleja los tópicos más investigados y discutidos en la muestra de la literatura revisada.

Se encontró que algunos autores les proporcionan a los ingenieros de pruebas una base teórica y práctica para una implementación exitosa de la automatización [24-27]; otros, con amplia experiencia profesional en la industria, los ayudan en la decisión de automatizar las pruebas, a navegar a través de una gran cantidad de herramientas para seleccionar las que mejor se ajusten, y les dan consejos sobre la construcción de procesos de pruebas sólidos y documentados [26]; también ofrecen orientación sobre planificación, diseño, desarrollo, ejecución y evaluación de las pruebas [24, 27].

Existe algunas razones para la automatización, y las organizaciones que la aplican piensan en cuestiones tales como mejorar tiempos de entrega y mantener pruebas repetibles [19], ahorrar recursos [28], o gestionarlas en menos tiempo [29]. Si se implementa bien, la automatización puede ser un medio eficaz para reducir el esfuerzo en el desarrollo, porque se puede eliminar o minimizar tareas repetitivas y reducir el riesgo de introducir errores humanos [30]. Las pruebas automatizadas pueden ser consideradas como una piedra angular en el proceso de desarrollo de software, donde su uso está aumentando rápidamente debido a la introducción de métodos como el desarrollo basado en pruebas [31] y la integración continua [32].

Las pruebas automatizadas son dependientes de una herramienta, ya sea para el flujo de prueba del entorno de configuración, la ejecución de la prueba o el análisis de resultados para su desmonte, por esto es necesario extraer el máximo beneficios de ellas [33, 34]. En consecuencia, la complejidad y el tamaño total de la prueba comúnmente son del orden de, o mayores que, la complejidad y el tamaño del producto bajo prueba [35]. Si esta complejidad no se administra adecuadamente puede generar contratiempos costosos para la automatización, inclusive llevar a su abandono [36]. El propósito de realizar medición a la automatización es permitir que el probador desarrolle sus habilidades para llevar a cabo esta tarea, o ejecutarla de manera más eficiente que si se hiciera manual, y si esto falla entonces la automatización no tiene valor [19].

Los problemas que se solucionan con software son cada vez más complejos, por lo que la automatización juega un papel importante cuando se presentan cambios o mejoras, y para entregar un producto con alta calidad y a tiempo [37]. En esa entrega hay tres aspectos clave: la velocidad, la calidad y la incertidumbre, que prácticamente son el mantra de la Era de la Información [38]. Esto implica que la velocidad de las nuevas funcionalidades decidirá el grado de satisfacción de los clientes, así como de mantenerlos y de atraer nuevos [25].

Además, en términos de desarrollo de software tradicionalmente la prueba se considera como el último paso antes de ir a producción, y a menudo el tiempo que se deja para hacerla no es suficiente, especialmente si se ejecuta manualmente [39]. Como resultado, la liberación del producto puede verse afectada [40]. El aumento de la complejidad de los sistemas significa que hay un número potencialmente indefinido de combinaciones de datos de entrada, que dan lugar a salidas diferentes y a que a menudo muchas de ellas no seann cubiertas por las pruebas manuales [41].

Los beneficios de la automatización están soportados por las evidencias de los trabajos analizados, pero en comparación con los experimentos y los estudios de casos, aparecen solamente unos pocos informes de experiencias reportadas [25, 36, 42]. Los beneficios más documentados son:

- Mejora la calidad del producto, en términos de un menor número de defectos presentes en el ciclo de vida [30, 43].
- Incrementa y mejora la cobertura de la prueba y del código, y hay mayor cantidad de valores de entrada probados [30, 43-47].
- Reduce el tiempo de la prueba, es decir, ofrece la capacidad de ejecutar más pruebas en un lapso menor [25, 30, 48, 49].
- Mejora la fiabilidad, porque maneja la repetición de acuerdo con el conocimiento adquirido por los probadores [25].
- Incrementa la confianza, por ejemplo, desde la percepción de los desarrolladores y los usuarios [1, 25].
- Reutiliza casos de prueba, porque se diseñan con el mantenimiento en mente, y un alto grado de repetición trae beneficios [30, 50].
- Reduce el esfuerzo humano, lo que se puede aprovechar en otras actividades [36, 48, 50-52].
- Disminuye costos, porque la automatización no requiere mucha intervención humana [45, 50, 53].
- Aumenta la detección de fallos, con lo que su eficacia es superior a las pruebas manuales [42, 47, 53-56].

Por su parte, Fewster y Graham [25] afirman que uno de los beneficios más importantes de la automatización es que se puede ejecutar y repetir muchos casos de prueba, facilita el uso de herramientas de cobertura e incrementa significativamente el refinamiento iterativo de los casos a aplicar. Además, ayuda a crear mejores casos de prueba, con una calidad definida y con menos esfuerzo [36]. Otros autores vinculan los beneficios en diversos sentidos, como que al utilizar herramientas especiales se puede reducir el esfuerzo y los costos relacionados con la prueba [51], u opinan que la prueba exhaustiva es imposible por lo que alcanzar una alta cobertura sigue siendo una expectativa, y con la automatización se logra avanzar en su consecución [19].

La automatización puede reducir el costo del fracaso, al permitir una mayor cobertura para que los errores sean descubiertos antes de que tengan la oportunidad de causar un daño real en el producto [38]. A diferencia de los beneficios, las limitaciones de la automatización se basan principalmente en informes de experiencias y en unos cuantos estudios empíricos [30, 57]. Algunas limitaciones reportadas son:

- La automatización no reemplaza totalmente las pruebas manuales, especialmente las que requieren un amplio conocimiento del dominio [44, 52].
- Problemas para alcanzar los objetivos esperados, porque al ejecutar las pruebas en una fracción de tiempo no se puede alcanzar beneficios duraderos o reales [28, 57].
- Dificultades en el mantenimiento, principalmente debido a los cambios en las tecnologías y la evolución de los productos software [58].
- El proceso necesita tiempo para madurar, porque crear la infraestructura y las pruebas requiere tiempo, por lo tanto, la madurez de la automatización y otras prestaciones conexas también [42].
- Crea falsas expectativas, debido a que las organizaciones no tienen claridad suficiente y su objetivo es ahorrar la mayor cantidad de tiempo y dinero que sea posible [51, 59].
- Estrategias inadecuadas, porque son difíciles de decidir, por lo tanto, no se alcanza el beneficio de la automatización [51, 58].
- Falta de personal cualificado, porque los ingenieros necesitan desarrollar buenas habilidades para implementar la automatización [19].

Automatizar las pruebas puede generar muchos beneficios, pero puede que no sea fácil de implementar, porque hay que considerar la realización de cambios en los procesos de la estructura de la organización [60]. La iniciativa puede parecer solamente un cambio de lo manual a lo automático, sin embargo, idealmente debe ser considerada en términos de todos los factores posibles que pueden verse afectados por los cambios, tales como las personas, la tecnología y los procesos [13]. Desde la perspectiva de las personas hay que centrarse en cómo hacer que los usuarios acepten los cambios, en lugar de forzarlos a implementarlos, porque la adopción de una nueva tecnología no es una tarea fácil y es un proceso de la cognición del usuario que requiere tiempo [13].

En este sentido, los investigadores han estudiado por largo tiempo cómo y por qué las personas adoptan nuevas tecnologías, y se ha conformado varias corrientes [9]. Algunas se centran en la aceptación individual mediante la intención de uso o el uso en sí como una variable dependiente [61]. Desde esta perspectiva hay que pensar en las limitaciones hacia el uso de la automatización, y la mayoría de organizaciones tienen su propio sistema interno que, después de años de acumulación, se ha vuelto cada vez más complejo [13].

Otra cuestión que se analiza en los trabajos es si la automatización puede sustituir la prueba manual, y uno de los puntos de vista propone que eso no es posible, porque las pruebas manuales detectan más nuevos defectos que las automatizadas [36]. También opinan que entre el 60% y el 80% de los errores encontrados durante un esfuerzo de prueba automatizada ya se han logrado manualmente, y que a menos que desde el principio se cree y ejecute nuevos casos de prueba en la herramienta, la mayoría de los errores se encuentra manualmente [62]. Estos dos puntos de vista probablemente impliquen que la automatización puede ser utilizada como una especie de herramienta suplementaria, pero no como un reemplazo completo de la prueba manual.

Mientras que estos autores visualizan a la automatización como una herramienta complementaria, otros se preocupan por cómo encajarla en un proceso en curso. Esto exige que el proceso aplicado debe ser lo suficientemente maduro, porque si es deficiente la automatización no ayudará [40]. La automatización necesita tiempo para madurar, lo mismo que para crear la infraestructura necesaria, por lo tanto, la madurez de la automatización, y otros conexos, requiere tiempo [19]. Otros autores afirman que para la automatización se necesita personas con habilidades adecuadas, porque implementarla no significa que los probadores van a sobrar, y si no tienen las habilidades requeridas se les debe proporcionar formación [15]. Fecko y Lott [42] proponen algunos retos para la automatización de las pruebas. Uno de ellos indica que hay que enseñarles a los probadores cómo escribir y ejecutar pruebas automatizadas, como parte regular de sus responsabilidades.

Dascal y Dror [63] afirman que los avances tecnológicos, como el software, han sido una parte integral del desarrollo humano a lo largo de la historia, y que los grandes avances de las últimas décadas han hecho que muchos penetren rápidamente en la vida cotidiana. Por lo que las pruebas automatizadas se han aplicado en algunos durante años, sin embargo, todavía son un concepto nuevo para la mayoría de las organizaciones [7]. Como una tecnología nueva la comprensión de la automatización es importante, porque es un paso indispensable para su adopción, y algunas estimaciones indican que, desde la década de 1980, alrededor del 50% de las nuevas inversiones en las organizaciones ha sido en TI [61]. Aun así, para que esas tecnologías mejoren la productividad primero deben ser comprendidas, aceptadas y utilizadas, un proceso lento en el caso de la automatización [61]. Aparte de los puntos relativos al rendimiento y la calidad del trabajo, también se hace hincapié en que para la aceptación de la automatización es necesario disfrutarla [64], porque los empleados con poco interés pueden conducir a una menor productividad y entorpecer su adopción [65].

Para Bertolino [15] la automatización es una forma de mantener la calidad del análisis y las pruebas, en línea con la cada vez mayor cantidad y complejidad del software. De acuerdo con esta autora, la investigación actual en Ingeniería del Software enfatiza en la automatización de la producción de software, con una generación de herramientas de desarrollo más complejas que producen mayor cantidad de código con menos esfuerzo. Pero la otra cara de la moneda es el peligro de que los métodos para evaluar la calidad de esa cantidad de software, los métodos de prueba en particular, no puedan mantener ese ritmo de construcción, por lo que Berner et al. [36] afirman que la investigación actual debe tener por objeto mejorar lo más posible el grado de la automatización, ya sea mediante el desarrollo de técnicas avanzadas para generar pruebas, o mediante la búsqueda de procedimientos de soporte innovadores para automatizar el proceso.

El sueño con la automatización es que pueda desarrollar por sí misma un entorno de prueba integrado, y que como una pieza de software se complete, se despliegue y se encargue automáticamente de reparar y generar o recuperar el código defectuoso, produciendo casos de prueba adecuados, ejecutarlos y finalmente emitir un informe de resultados [66]. Esta idea, aunque quimérica, ha atraído a algunos investigadores [67, 68] con el objetivo precisamente de ejecutar pruebas automáticas, al mismo tiempo que se desarrolla el programa. Otros trabajan en la noción de la agitación del software [69], una técnica de pruebas unitarias automáticas apoyada en la herramienta comercial Agitator, que combina diferentes análisis tales como la ejecución simbólica y la resolución de restricciones, y que genera entradas aleatorias de datos con el apoyo del sistema Daikon [70].

Por su parte, Dustin y Garrett [27] presentan un debate constructivo sobre qué casos de prueba deben ser automatizados y sobre directrices para evaluar el rendimiento de la inversión, un tema

que también tratan Mosley y Posey [26]. Además, argumentan en contra de la idea de un ciclo de vida de las pruebas del software, y afirman que el resultado de la automatización depende de la calidad de los procesos ya existentes en la organización. Dustin et al. [24] introducen una metodología estructurada orientada a asegurar la implementación exitosa de las pruebas automatizadas, y promueven que debe ser una preparación deliberada y bien razonada, incluyendo estudios en profundidad de los requisitos de prueba, y estableciendo expectativas realistas y una planificación estructurada.

4. ANÁLISIS DE RESULTADOS

De acuerdo con muchos de los autores analizados en esta investigación, la automatización de las pruebas es una de las herramientas más eficaces para reducir los costos y el tiempo de un plan de pruebas. Desde cuando aparecieron las primeras herramientas a mediados de 1980, el proceso se implementó en diferentes tipos de pruebas, y en poco tiempo se convirtió en centro de atención, tanto por sus ventajas como por los problemas que puede causar, desde el aumento de los costos de producción hasta la disminución de la fiabilidad e incluso el fracaso de los proyectos. Sin embargo, su popularidad se ha incrementado constantemente debido a las ventajas involucradas: reduce la participación de los probadores, lo que significa una disminución en los errores humanos; disminuye el tiempo de las pruebas, y por lo tanto el costo del producto final; y la comprobación automática es unas 25 veces más rápida que la manual, por lo que los errores se descubren más rápido y se atienden igualmente.

Por otra parte, la mayoría de personas se imagina al proceso de prueba como una secuencia de acciones, y de hecho se puede describir como una secuencia de interacciones intercaladas con evaluaciones. El asunto es que las interacciones se pueden predecir, pero la mayoría son mal especificadas, complejas y ambiguas. Sin embargo, si el objetivo principal es reducir la prueba a un conjunto de acciones rutinarias, los enfoques para conceptualizar una secuencia general de acciones de prueba pueden ser útiles. Pero incluso en esta situación el resultado puede ser superficial y limitado. Por otra parte, la prueba manual tiene la propiedad de adaptarse, lo que significa que puede cambiar fácilmente de acuerdo con las circunstancias de cada contexto.

Por lo tanto, las personas no requieren una estricta secuencia de acciones para revelar muchos defectos y para distinguir anomalías inofensivas, lo que constituye una gran ventaja comparada con la automatización. Es decir, la automatización es la mejor opción para un reducido espectro de pruebas. Otros afirman que un error común es creer que probar significa repetir las mismas acciones una y otra vez. Aunque la realidad es que, si en la primera ejecución del caso de prueba no se determina ningún error, los existentes solamente podrán revelarse mediante otras ejecuciones, pero en otro contexto del plan de pruebas. En este sentido las pruebas manuales tienen una amplia variedad de casos de prueba, y proporcionan una buena tasa de éxito en la detección de errores nuevos y viejos. Por lo tanto, los casos de prueba ejecutados tendrán buenos resultados solamente si son variados.

En contra de muchas opiniones generalizadas no todas las acciones de prueba se pueden automatizar. Algunas tareas son muy fáciles para los seres humanos, pero al mismo tiempo también lo son para los computadores. Por eso es que la parte más difícil de la automatización es la interpretación de los resultados de la prueba, además, porque el desarrollo de software es innovador, lo que significa que tiene un alto grado de incertidumbre que agrava el problema de la automatización. Asimismo, un proyecto software se desarrolla utilizando un proceso incremental que involucra diferentes tipos de componentes, incluso en las últimas fases del siclo de vida, lo que tampoco colabora con la automatización.

Por lo tanto, las pruebas automatizadas se pueden transformar fácilmente en un proceso lento, costoso e ineficaz, y esto contradice las opiniones de que una prueba automatizada es más rápida, porque no necesita intervención humana. Esta expresión es incorrecta, no solamente porque el proceso puede ser más lento, sino que siempre requiere intervención humana, porque el proceso de analizar los resultados y corregir los errores lo deben llevar a cabo personas. De la misma forma que es imposible imaginar una prueba sin contratiempos, igualmente es imposible imaginar la automatización sin intervención humana.

Una corriente analizada afirma que la automatización reduce los errores humanos, y por eso es la decisión perfecta para implementar una larga lista de acciones seculares que son difíciles para las personas. Esto es que alienta a la mayoría de directores de TI cuando cuantifican los costos y los beneficios de las pruebas automatizadas, al compararlas con las manuales después de implementar la nueva estrategia. Por desgracia, estos procesos son muy diferentes y cada tipo de prueba tiene diferentes conceptos y dinámicas que se estructuran para determinar diferentes tipos de errores.

Por lo tanto, no tiene sentido una comparación directa en términos de costos por número de errores encontrados. Otra expectativa que despierta la automatización es el ahorro significativo de costos laborales. La realidad es que el valor de las pruebas automatizadas tiene varios componentes: desarrollo, operación, mantenimiento, cambios en los productos y de otras tareas necesarias, y originadas por la automatización. Al compararlos con el valor total de las acciones de la prueba manual, no se reduce significativamente, porque no disminuye el trabajo de los probadores hasta el punto de que no se requieran.

El costo del trabajo de los probadores depende de muchos factores, incluyendo el de las herramientas utilizadas, su destreza, la calidad del conjunto de casos de prueba y muchos otros diferentes. Obviamente, escribir una sola prueba implica menos esfuerzo que de la de un sistema grande que incluya la elección de las herramientas, la coordinación de la automatización y otras acciones. Esto puede tomar meses de trabajo duro. Los costos de la automatización también incluyen los de las nuevas tareas especiales que requiere, por lo tanto, estas pruebas son una acción costosa que incluye cargos por la documentación de los casos de prueba, el proceso, el control y análisis de resultados, las modificaciones al código, la organización del proceso de desarrollo, y todos los que necesitan la participación de personas.

Para los autores que piensan de esta forma la automatización no es una buena herramienta para reducir costos laborales. Varios investigadores opinan que una cuestión que no se toma muy en serio es la probabilidad de dañar el proyecto de pruebas con la automatización. Por desgracia, el diseño y la implementación de una automatización equivocada pueden causar un alto número de problemas. Por eso se requiere una comprensión plena de sus posibilidades de rendimiento antes de tomar la decisión de implementarla. Cuando no se hace esto la automatización podría transformarse en una gran cantidad de código de prueba, que los probadores no entienden completamente, porque no hay estrategias claras.

En consecuencia y como resultado del análisis a los resultados de esta investigación, se puede decir que en la literatura se encuentra opiniones y resultados diversos acerca de la automatización de las pruebas. En términos generales se pueden resumir en que:

- Las pruebas automatizadas no son una simple secuencia de comandos, sino una sucesión de interacciones entremezcladas con evaluaciones.

- Automatizar no significa repetir las mismas acciones una y otra vez, sino que se requiere variaciones continuas.
- No todos los tipos de prueba se pueden automatizar.
- La prueba automatizada reduce los errores de acción del probador, pero no reduce los errores del diseño.
- La prueba automatizada no sustituye completamente a la prueba manual, sino que la complementa.
- La automatización disminuye el valor del proceso de pruebas, pero aumenta el costo de desarrollo de las pruebas.
- Un plan de pruebas automatizadas requiere una documentación completa, para lo cual tradicionalmente no se aparta el tiempo suficiente.
- La prueba manual encuentra más defectos que una automatizada.
- La automatización mejora el uso de recursos y brinda confianza en la prueba de atributos, que manualmente son difíciles.
- La corrección de los resultados esperados tiene mayor dependencia.
- La automatización no mejora la eficacia de la prueba.
- La calidad de la automatización es independiente de la calidad de la prueba.
- Las herramientas de prueba no son muy flexibles.
- La automatización puede aumentar significativamente la productividad y calidad de la prueba.
- La automatización mecaniza los pasos de las pruebas manuales utilizando herramientas.
- Con la automatización se incrementa la velocidad de ejecución del plan de pruebas y es más confiable, repetible, programable, integral y reutilizable.
- La automatización no soluciona todos los problemas de las pruebas manuales.

5. CONCLUSIONES

Debido a que las tácticas de la Ingeniería del Software se mueven hacia estándares más abiertos y se centran en la reducción del tiempo de desarrollo y en la creación de capacidades reutilizables, la necesidad de pruebas eficaces y exhaustivas se vuelve más importante que nunca. Los artefactos de prueba ya no pueden ser usados solamente durante el desarrollo inicial y luego desechados. Construir software incremental de forma coordinada, y con capacidades de prueba altamente reutilizables, les permitirá a los desarrolladores reutilizar casos de prueba y herramientas. Las pruebas de software automatizadas utilizan herramientas para ejecutar el software, medir las respuestas y evaluar su exactitud sin intervención humana significativa.

Aunque se ha progresado bastante y la ciencia y la academia difunden resultados continuamente, de acuerdo con Serna et al. [71], y los resultados de esta investigación, la automatización de las pruebas se encuentra todavía en una etapa de experimentación. Esta investigación revela que presenta ventajas a la vez que limitaciones, y los autores difunden resultados especialmente de experiencias no comprobadas. Vale la pena notar que uno de los principales motivos para automatizar es ahorrar tiempo, porque la prueba se continúa ejecutando como una fase al final del ciclo de vida, cuando ya no se tiene tiempo suficiente para una prueba exhaustiva. Por otro lado, parece que para los clientes es apropiado exigir requisitos en forma de pruebas de aceptación automatizadas. Pero esto necesariamente no refleja la viabilidad de la automatización, solamente es una cuestión de cómo darle prioridad y satisfacer al cliente.

Si bien la automatización de las pruebas tiene un gran potencial para mejorar la eficiencia y la calidad de los productos software, no menos importante es el mantenimiento de este tipo de pruebas y los costos asociados. Esto hace que sea importante considerar cuidadosamente y por adelantado el potencial del beneficio versus el costo. En correspondencia, con que una prueba no descubra errores puede ser engañosa, y automatizar todo el plan de pruebas puede crear una falsa sensación de control del proceso, porque parece ser que automatizar aumenta el nivel de confianza entre los desarrolladores, los probadores y el cliente. La base de experiencias limitada en la automatización encontrada en esta investigación demuestra que se ha alcanzado algunos logros importantes, sin embargo, todavía hay necesidad de investigar en esta práctica. En este capítulo se ha señalado algunos temas y conceptos sobre los que viene trabajando, pero los resultados no son muy alentadores para llegar a una automatización total en corto tiempo.

La demanda por el incremento de la funcionalidad del software, de un desarrollo rápido y de la disminución de los costos de producción, sin comprometer la calidad, no deja otra opción que trabajar arduamente por automatizar totalmente las pruebas del software. Para algunos autores esta estrategia puede ser sorprendentemente eficaz y altamente beneficiosa para cualquier organización, pero, aunque el costo de configuración inicial puede ser alto, el rápido retorno de la inversión proyectado lo supera, y produce beneficios clave para incrementar la productividad, la disponibilidad, la fiabilidad y el rendimiento de cualquier empresa. Las pruebas automatizadas son particularmente eficaces cuando la naturaleza del trabajo es repetitivo y rutinario, tales como las pruebas unitarias y las de regresión. Otros trabajos informan que usar las pruebas automatizadas les permitió probar grandes volúmenes de código, lo que habría sido imposible de otra manera. En todo caso, y dado que no hay uniformidad acerca de las ventajas de la automatización, se espera que en el futuro cumpla con lo prometido desde hace décadas.

REFERENCIAS

[1] Haugset B. y Hanssen G. (2008). Automated acceptance testing: A literature review and an industrial case study. En Agile Development Conference. Toronto, Canada.

[2] Huang L. y Holcombe M. (2008). Empirical investigation towards the effectiveness of test first programming. Information and Software Technology 52(1), 182-194.

[3] Safronau V. y Turlo V. (2011). Dealing with challenges of automating test execution. En Third International Conference on Advances in System Testing and Validation Lifecycle. Barcelona, Spain.

[4] Hoffman D. (1999). Test automation architectures: Planning for test automation. En 12th International Software Quality Week. San Francisco, USA.

[5] Thummalapenta S. et al. (2011). Automating test automation. IBM Research Report.

[6] Sommerville I. (2010). Software Engineering. Person.

[7] Askarunisa A. et al. (2009). Selecting effective coverage testing tool based on metrics. The ICFAI University Journal of Computer Sciences 3(3), 47-70.

[8] Abdel T. (1988). The economics of software quality assurance: A simulation-based case study. MIS Quarterly 12(3), 395-411.

[9] Feldman F. (2005). Quality assurance: Much more than testing. Queue – Quality Assurance 3(1), 26-29.

[10] Serna E. y Arango F. (2011). Software testing: More than a stage in the life cycle. Revista de Ingeniería 35, 34-40.

[11] Serna E. et al. (2015). Software testing is more than an emergency plan. En 2nd International Conference on Communication Technology. Melbourne, Australia.

[12] Serna E. (2013). Prueba funcional del software - Un proceso de Verificación constante. Fondo Edi. ITM.

[13] Stobie K. (2005). Too darned big to test. Queue – Quality Assurance 3(1), 30-37.

[14] Koochakzadeh N. y Garousi V. (2010). A tester-assisted methodology for test redundancy detection. Advances in Software Engineering, Article 6.

[15] Bertolino A. (2007). Software testing research: Achievement, challenges, dreams. En Future of Software Engineering Conferenne. Minneapolis, USA.

[16] Dustin E. et al. (2000). Automated Software Testing. Addison-Wesley.
[17] Bavin P. (2010). A proposal for a repertoire of software testing methods. Master's Thesis. Lappeenranta University of Technology.
[18] Whyte G. y Mulder D. (2011). Mitigating the impact of software test constraints on software testing effectiveness. The Electronic Journal Information Systems Evaluation 14(2), 254-270.
[19] Rafi D. et al. (2012). Benefit and limitations of automated software testing: Systematic literature review and practitioner survey. En 7th International Workshop on Automation of Software Test. Zurich, Switzer.
[20] Majikes J. et al. (2013). Literature review of testing techniques for medical device software. En Medical Cyber Physical Systems Workshop. Philadelphia, USA.
[21] Wiklund K. et al. (2014). Impediments for automated testing - An empirical analysis of a user support discussion board. En Seventh Inter. Conf. on Soft. Testing, Verification and Validation. Cleveland, USA.
[22] Guan D. (2014). Manual to automated testing. Master's Thesis. Victoria University of Wellington.
[23] Torraco R. (2005). Writing integrative literature reviews: Guidelines and examples. Human Resource Development Review 4(3), 356-367.
[24] Dustin E. et al. (1999). Automated software testing: Introduction, management, and performance. Addison Wesley.
[25] Fewster M. y Graham D. (1999). Software test automation: Effective use of test execution tools. Addison Wesley.
[26] Mosley D. y Posey B. (2002). Just enough test automation. Prentice-Hall.
[27] Dustin E. y Garrett T. (2009). Implementing automated software testing: How to save time and lower costs while raising quality. Addison Wesley.
[28] Persson C. y Yilmaztürk N. (2004). Establishment of automated regression testing at ABB: Industrial experience report on 'avoiding the pitfalls'. En 19th IEEE International Conference on Automated Software Engineering. Linz, Austria.
[29] Taipale O. et al. (2011). Trade-off between automated and manual software testing. International Journal of System Assurance Engineering and Management 2(2), 114-125.
[30] Karhu K. et al. (2009). Empirical observations on software testing automation. En Second International Conference on Software Testing Verification and Validation. Denver, USA.
[31] Janzen D. y Saiedian H. (2005). Test-driven development concepts, taxonomy, and future direction. Computer 38(9), 43-50.
[32] Stolberg S. (2009). Enabling agile testing through continuous integration. En Agile Conference. Chicago, USA.
[33] Gallesdic I. y Killiospy G. (2013). Software testing as science. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 3(1), 33-37.
[34] Petrova E. et al. (2014). Automate or not to automate acceptance tests: Not is a dilemma. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 4(1), 30-33.
[35] Kasurinen J. et al. (2010). Software test automation in practice: Empirical observations. Advances in Software Engineering - Special issue on software test automation, Article 4.
[36] Berner S. et al. (2005). Observations and lessons learned from automated testing. En 27th International Conference on Software Engineering. St. Louis, USA.
[37] Hayes L. (1997). The truth about automated test tools. Datamation 43(4), 45.
[38] Hayes L. (1989). The automated testing handbook. Software Testing Institute.
[39] Davis F. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Quarterly 13(3), 319-340.
[40] Damm L. et al. (2005). Introducing test automation and test-driven development: An experience report. Electronic notes in theoretical computer science 116, 3-15.
[41] Blackburn M. et al. (2004). Why model-based test automation is different and what you should know to get started. En International Conference on Practical Software Quality Techniques y Testing Techniques. Washington, USA.
[42] Fecko M. y Lott C. (2002). Lessons learned from automating tests for an operations support system. Software: Practice and Experience 32(15), 1485-1506.
[43] Malekzadeh M. y Ainon R. (2010). An automatic test case generator for testing safety-critical software systems. En 2nd Inter. Conference on Computer and Automation Engineering. Singapore, Singapore.

[44] Tan R. y Edwards S. (2008). Evaluating automated unit testing in sulu. En First International Conference on Software Testing, Verification, and Validation. Lillehammer, Norway.
[45] Alshraideh M. (2012). A complete automation of unit testing for JavaScript programs. Journal of Computer Science 4(12), 1012-1019.
[46] Burnim J. y Sen K. (2008). Heuristics for scalable dynamic test generation. En 23rd IEEE/ACM International Conference on Automated Software Engineering. L'Aquila, Italy.
[47] Saglietti F. y Paint F. (2011). Automated unit and integration testing for component-based software systems. En Inter. Work. on Secu. and Depen. for Reso. Constrained Embedded Systems. Naples, Italy.
[48] Du Bousquet L. y Zuanon N. (1999). An overview of Lutess: A specification-based tool for testing synchronous software. En 14th Conference on Automated Software Engineering. Cocoa Beach, USA.
[49] Wissink T. y Amaro C. (2006). Successful test automation for software maintenance. En 22nd IEEE International Conference on Software Maintenance. Philadelphia, USA.
[50] Dallal J. (2009). Automation of object-oriented framework application testing. En 5th IEEE GCC Conference and Exhibition. Kuwait City, Kuwait.
[51] Pocatilu P. (2002). Automated software testing process. Economy Informatics 1, 97-99.
[52] Leitner A. et al. (2007). Reconciling manual and automated testing: The autotest experience. En 40th Hawaii International Conference on Systems Science. Waikoloa, USA.
[53] Shan L. y Zhu H. (2009). Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. Computing Journal 52(5), 571-588.
[54] Coelho R. et al. (2007). Jat: A test automation framework for multi-agent systems. En 23rd IEEE International Conference on Software Maintenance. Paris, France.
[55] Hao D. et al. (2009). Test-data generation guided by static defect detection. Journal of Computer Science and Technology 24(2), 284-293.
[56] Swain R. et al. (2012). Automatic test case generation from UML state chart diagram. International Journal of Computer Applications 42(7), 26-36.
[57] Bashir M. y Banuri S. (2008). Automated model based software test data generation system. En 4th International Conference on Emerging Technologies. Venice, Italy.
[58] Liu C. (2000). Platform-independent and tool-neutral test descriptions for automated software testing. En 22nd International Conference on Software Engineering. Limerick, Ireland.
[59] Pettichord B. (1999). Seven steps to test automation success. En STAR West Conference. San Jose, USA.
[60] Polo M. et al. (2007). Integrating techniques and tools for testing automation. Software Testing, Verification and Reliability 17(1), 3-39.
[61] Venkatesh V. et al. (2003). User acceptance of information technology: Toward a unified view. MIS Quarterly 27(3), 425-478.
[62] Kaner C. (1997). Improving the maintainability of automated test suites. En 10th International Software Quality Week. San Francisco, USA.
[63] Dascal M. y Dror I. (2005). The impact of cognitive technologies. Pragmatics y Cognition 13(3), 451-457.
[64] Prusch A. et al. (2011). Integrating technology to improve medication administration. American journal of health-system pharmacy 68(9), 835-842.
[65] Attar G. y Sweis R. (2010). The relationship between information technology adoption and job satisfaction in contracting companies in Jordan. Journal of infor. techn. in construction 15, 44-63.
[66] Östrand T. et al. (2005). Predicting the location and number of faults in large software systems. IEEE Transactions Software Engineering 31(4), 340-355.
[67] Saff D. y Ernst M. (2004). An experimental evaluation of continuous testing during development. En International symposium on Software testing and analysis. Boston, USA.
[68] Briand L. et al. (2006). Automated, contract-based user testing of commercial-off-the-shelf components. En 28th International Conference on Software Engineering. Shanghai, China.
[69] Boshernitsan M. et al. (2006). From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. En International symposium on Software testing and analysis. Portland, USA.
[70] Ernst M. et al. (2007). The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1-3), 35-45.
[71] Serna E. et al. (2015). Current maturity of development of software test automation. En II Congreso Internacional de Ingeniería. Trujillo, Perú.

CAPÍTULO XVIII
Un modelo para determinar la madurez de la automatización de las pruebas del software como área de investigación y desarrollo[1]

Edgar Serna M.[1]
Raquel Martínez M.[2]
Paula Tamayo O.[2]
[1]Instituto Antioqueño de Investigación
[2]Institución Universitaria de Envigado

En este capítulo se propone un modelo para determinar la madurez actual de la automatización de las pruebas como área de investigación y de desarrollo en la industria del software. Se describe el proceso y los resultados de una investigación, que tiene como objetivo determinar el nivel de madurez de la automatización. Se realizó una revisión sistemática de la literatura en la que se encontraron 978 trabajos que describen modelos de madurez de las pruebas del software, y/o analizan la eficacia y la eficiencia de la automatización. Luego de un proceso de análisis y de aplicar los criterios de inclusión/exclusión y de valoración a la calidad, se extrajeron 26 trabajos. La conclusión final es que el nivel de madurez actual de la automatización de las pruebas del software es *Adolescente*.

[1] Publicado en la Revista Computación y Sistemas 21(2), 337–352. 2017.

INTRODUCCIÓN

La sociedad de este siglo es software-dependiente, porque este desarrollo tecnológico tiene cada vez más un fuerte impacto en las operaciones vitales de la vida humana, tales como la medicina, la aeronáutica, la investigación espacial, las telecomunicaciones y la protección de datos, entre otras [1]. Por eso es imperativo abordar los problemas de calidad relacionados tanto con el proceso del desarrollo de software como con el producto mismo. Esta investigación se centra en el proceso y a analizar el nivel de madurez que ha alcanzado la automatización de las pruebas.

En un sentido amplio la prueba se aplica para cubrir todas las actividades relacionadas con la calidad del software por lo que, si se mejora el proceso y la aplicación de criterios de madurez, se logrará un impacto positivo en la productividad de la Ingeniería del Software, y se reducirá los esfuerzos y el tiempo de producción [2].

Por otro lado, y debido a que los problemas son cada vez más complejos y a que los clientes son más exigentes sobre la calidad de los productos, la prueba es una actividad esencial en el desarrollo de software. Es necesario probar para minimizar el riesgo de entregar sistemas con fallas, y la automatización de las pruebas surgió como una alternativa para probar en menos tiempo un ámbito más amplio de sus funcionalidades. En términos generales consiste en utilizar software para ejecutar o apoyar las actividades de prueba, tales como la gestión, los casos de prueba, la ejecución y la evaluación de resultados. Sin embargo, la industria todavía la concibe como una actividad para incrementar la productividad del equipo, a la vez que la calidad de los productos [3].

Otra cuestión relevante para incrementar la calidad en el desarrollo de software es utilizar modelos de madurez para apoyar el mejoramiento continuo de los procesos del ciclo de vida. Pero, aunque existe diversos modelos que cubren este ámbito de actividades, hay otros que se construyen específicamente para desarrollar la cultura de las pruebas y, por lo tanto, soportar su introducción de forma más disciplinada y programada. Entre todas estas propuestas nadie puede discutir que la automatización de las pruebas hace parte del abanico para lograrlo. Pero a las organizaciones, que buscan introducirla en sus procesos, no les es fácil seleccionar un modelo que les ayude a comprender cuáles son las mejores prácticas de la automatización y cómo introducirlas en contexto [3]. Las organizaciones que deciden aplicar esta tecnología pueden lograr uno de dos objetivos: ahorrar tiempo y dinero en lo cotidiano de la prueba, o no alcanzar este ahorro, pero sí mejoran la calidad del software más rápidamente que con las manuales [4].

Cuando la automatización alcance un adecuado nivel de madurez será posible aplicar las pruebas solamente con el toque de un botón, a la vez que seleccionar el mejor momento para hacerlo. Además, serán repetibles y utilizarán exactamente las mismas entradas en la misma secuencia de tiempo, algo que no se puede garantizar con las manuales. Pero, aunque pueda sonar sorprendente, probar es una habilidad, y para cualquier sistema existe un número astronómico de posibles casos de prueba, aunque solamente se tiene tiempo para correr un pequeño número de ellos. Sin embargo, se espera que con ello se encuentre la mayoría de los defectos en el software, por lo que hay que ser hábil para seleccionar y generar los casos de prueba más importantes. En este sentido, la experiencia ha demostrado por décadas que la selección al azar no es un método eficaz, por lo que se requiere un enfoque más reflexivo para lograrlo [5].

En este trabajo se propone un modelo para determinar la madurez de la automatización de las pruebas del software, a partir de una revisión a los que se ha publicado en la literatura. La propuesta se sustenta en el hecho de que los modelos analizados tienen objetivos diversos y no

logran determinar el nivel alcanzado por la automatización como área de investigación y desarrollo. Esto es importante para determinar en qué nivel se encuentra en las empresas, y colaborar para incrementarlo y mejorarlo para su beneficio. Se necesita conocer esa madurez, porque de nada le sirve a una organización esta tecnología si no tiene los argumentos suficientes para implementarla y obtener todos sus beneficios.

1. MARCO REFERENCIAL

Probar es aplicar una serie de actividades con el objetivo de descubrir y/o evaluar las propiedades de cada elemento del software [6]. Estas actividades pueden incluir planificación, preparación, ejecución, presentación de informes y gestión. Para Meyers [2] las pruebas son el proceso de ejecución de un programa con la intención de encontrarle errores, y Hass [7] opina que se puede considerar como una actividad de soporte, porque no tiene sentido sin los procesos de desarrollo, y porque no produce nada en sí misma: si no hay nada desarrollado no hay nada que probar.

Estas afirmaciones ofrecen una idea general de la definición de las pruebas de software y esencialmente conducen al objetivo general de las mismas: no se trata de encontrar todos los errores que pueda tener el sistema, sino de descubrir situaciones que podrían afectar negativamente su funcionamiento [8, 9]. Sin embargo, hay que tener en cuenta que el costo de encontrar y corregir errores puede elevarse considerablemente durante el ciclo de vida. Por lo tanto, cuanto antes se descubran los errores será mejor para controlar sus efectos, moderados o graves, en etapas posteriores.

La historia de las pruebas refleja la propia evolución del desarrollo de software que, por mucho tiempo, se focalizó en grandes programas científicos y militares, y en sistemas de bases de datos, producidos en plataformas mainframes o mini-computadores. Los escenarios de prueba se escribían en papel y las pruebas se orientaban a seguir las trayectorias de los flujos de control, al cálculo de algoritmos complejos y a la manipulación de datos. Un conjunto finito de procedimientos de prueba podía probar con eficacia un sistema completo, y el proceso generalmente se iniciaba hasta el final de la programación y lo ejecutaba el personal que estuviese disponible en el momento. Con el surgimiento del computador personal se iniciaron procesos de estandarización en toda la industria y en cómo desarrollar las aplicaciones software, para operacionalizarlas bajo un sistema operativo común.

La introducción del PC dio origen a una nueva era y generó un crecimiento explosivo del desarrollo de software comercial, y las aplicaciones empezaron a competir ferozmente por la supremacía y la supervivencia. Los productos líderes empezaron a rivalizar con calidad y eficiencia para satisfacer a los usuarios, y las metodologías de desarrollo evolucionaron aceleradamente. El esfuerzo de la prueba en estas nuevas metodologías requirió un enfoque diferente para probar el diseño, porque los flujos de trabajo podían ser llamados en casi cualquier orden. Esta capacidad exigía un alto número de procedimientos para soportar un sinfín de permutaciones y combinaciones.

Más tarde, la popularidad de las aplicaciones cliente-servidor introdujo una nueva complejidad en el esfuerzo de la prueba, porque el probador ya no ejercitaba una única aplicación para un sistema individual cerrado. Esta arquitectura implicaba tres componentes separados: el servidor, el cliente y la red. Por otro lado, la conectividad inter-plataformas aumentó la posibilidad de aparición de fallas, y las pruebas se tuvieron que relacionar con el rendimiento del servidor y la red, así como con el rendimiento general y la funcionalidad del sistema a través de esos componentes. Con el uso generalizado de las aplicaciones GUI, la captura de pantallas y la reproducción de escenarios

se convirtieron en una forma atractiva para probar aplicaciones. Entonces, se introdujeron herramientas automatizadas para hacerlo y poco a poco se popularizaron [2].

Aunque los escenarios de prueba y los *scripts* se seguían escribiendo en alguna aplicación de procesamiento de texto, el uso de estas herramientas se incrementó. Al ampliarse la complejidad y el esfuerzo de la prueba se requirió mayor planificación, por lo que el personal encargado de aplicarla fue obligado familiarizarse más con el sistema y a cumplir con requisitos de formación más específicos, relacionados con las plataformas y redes involucradas. Actualmente, estas herramientas han madurado y ampliado su capacidad, a la vez que aparecen otras con fortalezas y nichos específicos [8]. Además, la prueba automatizada se ha convertido cada vez más en un ejercicio de programación, aunque todavía involucra las funciones tradicionales de gestión, tales como trazabilidad de requisitos, planificación, diseño y desarrollo de escenarios y de *scripts*.

La base en todo este proceso son los casos de prueba, que se describen mediante atributos que determinan su calidad. Tal vez, el más importante sea su eficacia para detectar defectos, pero también que sea reutilizable, es decir, que se pueda modificar fácilmente para probar más de una cosa, lo que reduce el número de necesario [10]. Además, debe ser económico para realizar, analizar y depurar errores, y ser evolutivo y emplear la cantidad mínima de esfuerzo en su aplicación. A menudo estos atributos se deben equilibrar uno con otro. Hoy se acepta que la habilidad para realizar pruebas no consiste solamente en asegurar que los casos de prueba encuentren muchos defectos, sino también en que estén bien diseñados para evitar costos y tiempos excesivos [11].

Una forma de alcanzarlo es a través de la automatización, considerada por los equipos de prueba como una opción importante [8]. Aunque algunas organizaciones han fracasado rotundamente en su esfuerzo por implementarla y han tenido que recurrir nuevamente a los procesos manuales, a otras les ha permitido producir mejor software e incrementar su calidad rápidamente. Para obtener beneficios con la automatización las pruebas deben ser cuidadosamente seleccionadas y aplicadas, porque la calidad de este proceso es independiente de la calidad de la prueba, y el hecho de que se aplique automática o manualmente no afecta ni su eficacia ni su evolución.

No importa lo inteligente que se planee la automatización o lo bien que se haga, si la prueba en sí no logra nada entonces el resultado final será una evidencia de que nada se logra al hacerlo de esa forma. Habitualmente, una vez implementada es mucho más económica, porque el costo de funcionamiento es una fracción de los esfuerzos necesarios para hacerlo manualmente, sin embargo, en general cuesta más crearla y mantenerla. De ahí la importancia de seleccionar inteligentemente el momento para automatizar, porque será más barato ponerlo en práctica a largo plazo [4]. Además, hay que pensar en el mantenimiento, porque la sola actualización de un conjunto de pruebas automatizado puede tener un alto costo.

Para que la automatización de las pruebas sea eficaz y eficiente hay que comenzar con una buena materia prima, es decir, un buen banco de pruebas, un conjunto de pruebas hábilmente diseñado por un probador con las destrezas suficientes. Posteriormente, hay que tener la habilidad para automatizarlo de tal manera que se pueda crear y mantener a un costo razonable. En resumen, es posible que las pruebas sean de buena o de mala calidad, pero en todo caso será la habilidad del probador lo que determine la calidad total de la misma. También es posible que la automatización tenga buena o mala calidad, pero será la habilidad del automatizador lo que determine lo fácil que será agregar nuevas pruebas automatizadas, a la vez que mantenerlas y obtener los beneficios [12].

Es de amplio conocimiento que la automatización de las pruebas consiste en utilizar software para realizar o soportar todo tipo de actividades de prueba, tales como gestión, diseño, ejecución y análisis de resultados [13]. A menudo, las pruebas automatizadas se consideran como la realización de pruebas sobre secuencias de comandos, en lugar de tener probadores que lo realicen manualmente [6]. Sin embargo, también es cierto que muchas tareas y actividades adicionales de prueba pueden ser apoyadas por herramientas basadas en software. En general, la actividad de la automatización supone utilizar herramientas y, de acuerdo con Hass [7], el propósito es ejecutar el mayor número posible de actividades no-creativas, repetitivas y aburridas, además de explotar la ventaja de esas herramientas para almacenar y organizar grandes cantidades de datos. En términos generales la automatización puede ayudar a resolver: 1) el trabajo repetitivo, 2) la lentitud de las pruebas manuales, y 3) la inseguridad de las pruebas manuales. Por otro lado, la introducción de la automatización debe aumentar la productividad, porque de otra manera no compensaría el costo [7].

2. TRABAJOS RELACIONADOS

Diversos investigadores han presentado revisiones acerca de la automatización de las pruebas del software con diversos resultados, aunque no tienen en cuenta el análisis a la madurez del proceso como área de investigación y desarrollo. El objetivo de Michael Grottke [14] fue desarrollar un nuevo y ampliado modelo estadístico, para predecir la fiabilidad de los programas software con base en la madurez de las pruebas. El modelo se implementa mediante un prototipo que les ayuda a las pequeñas empresas de software a predecir la confiabilidad de sus desarrollos y a probarlos de manera fácil y eficiente, con una precisión superior a la disponible. Aunque la propuesta no tuvo la aceptación suficiente, se rescata la amplia revisión y comparación que realiza a los modelos previos. Analiza la automatización de las pruebas desde una perspectiva estadística, pero no se orienta a encontrar la madurez del proceso, sino a analizar los modelos propuestos.

La idea de la investigación de Ron Swinkels [15] es averiguar lo que se puede aprender de otros modelos de mejoramiento de los procesos de prueba. Para materializarla presenta una comparación entre Test Maturity Model TMM y siete modelos de mejoramiento, con el objetivo de extraer prácticas importantes para desarrollar otro modelo. El resultado es una descripción y comparación desde los objetivos, estructuras, áreas clave del proceso y procedimientos de evaluación de los modelos. Los resultados sirvieron para proponer un modelo propio, orientado básicamente al mejoramiento de los procesos de prueba. Aunque presenta una matriz de comparación, el modelo base que utiliza no es el referente más importante para la industria, debido a que su objetivo no es encontrar el nivel de madurez de la automatización, ni siquiera de los modelos existentes.

Shrini Kulkarni [16] da cuenta de la histórica de los modelos de madurez para las pruebas del software y su relevancia en el estado actual y futuro de la industria. También presenta algunas ideas iniciales y pensamientos en torno a un nuevo marco que refleje el estado actual de la prueba, en el que destaca la habilidad del probador y la importancia del pensamiento cognitivo y la inteligencia humana, lo que considera como una desviación de los modelos anteriores que ignoran el aspecto humano de las pruebas. El autor concluye que hay necesidad de modificar la mirada a los modelos y hacerlos más pertinentes a como está progresando el mundo en este aspecto. Los modelos de madurez del futuro tienen que reconocer el aspecto humano de la prueba y resistir la tentación de elaborar diagramas de flujo para las actividades, reconociendo la importancia del pensamiento crítico. El autor hace un análisis comparativo a cuatro modelos de madurez, pero siempre orientado a los aspectos humanos de la prueba, no a conocer o analizar el nivel de madurez de la automatización en general.

Gustavo De Souza [17] afirma que las mejores prácticas en las pruebas de software contribuyen a mejorar la calidad, y a reducir el costo de los productos, mediante la reducción de los tiempos en las etapas de prueba y durante la implementación y el mantenimiento. Los modelos de madurez para el desarrollo de software se han utilizado a gran escala para aliviar estos problemas, pero todavía no tienen en cuenta las actividades relacionadas con las pruebas, por lo que se hizo necesario desarrollar modelos de madurez. El autor hace una comparación y valida la eficiencia de los modelos Test Improvement Model TIM, Test Process Improvement TPI y Test Maturity Model TMM, e intenta determinar su nivel de madurez. El objetivo fue encontrar una guía para ayudarles a las empresas a mejorar la calidad de sus productos. La matriz de comparación utilizada es amplia y los indicadores ajustados, pero el hecho de que solamente trabaje con tres modelos no le permite presentar una imagen amplia del nivel de madurez de los relacionados con las pruebas del software. Además, su objetivo es demostrar el nivel de madurez de la organización con respecto a la automatización, no de la automatización como área de investigación.

Muhammad Sulayman [18] presenta una revisión sistemática de la literatura para identificar y analizar los modelos y técnicas existentes que utilizan las PyMes Web. Después de aplicar los filtros establecidos seleccionó un total de 88 estudios, pero sorprendentemente una inspección más detallada reveló que solamente cuatro de ellos eran relevantes para el tema. El objetivo principal fue investigar modelos o técnicas específicas para el mejoramiento de los procesos software, pero no encontró ninguno definido a la medida para las PyMes Web. Aunque las métricas analizadas incluyen a los equipos de desarrollo y la satisfacción de los clientes, el aumento de la productividad, el cumplimiento de los estándares y la excelencia operativa general, no identifica modelos de automatización para las pruebas, en parte por el tamaño de las empresas analizadas y porque sus limitaciones presupuestales no les permiten adentrarse en este campo. Esta investigación no pudo determinar el nivel de madurez de la automatización en PyMes Web.

Para Christiane von Wangenheim et al. [19] el nivel de madurez de la evaluación y el mejoramiento del software, guiado por procesos basados en un modelo de capacidad/madurez, se ha consolidado en la práctica como un medio eficaz para mejorar los procedimientos de desarrollo en las organizaciones. Describe los resultados de una revisión sistemática de la literatura sobre los modelos que han evolucionado el desarrollado y la adaptación. Los resultados demuestran que existe gran variedad de modelos con tendencia a la especialización para dominios específicos, pero que la mayoría se concentra en el marco CMM/CMMI y la norma ISO/IEC 15504, con los inconvenientes subsecuentes. Aunque los autores analizan el nivel de madurez su objetivo es mostrar una matriz comparativa entre los 29 modelos evaluados, por lo que no presentan una evaluación a la madurez de la automatización como área de investigación y desarrollo.

El estudio de Heiskanen, Maunumaa y Katara [3] muestra que la introducción de esta tecnología no siempre se realiza con éxito, en parte porque los procesos de prueba y de la organización no siempre se ajustan adecuadamente. En este trabajo se presenta un modelo de generación de pruebas automatizadas sobre el modelo TPI y traza un perfil de base para la introducción exitosa de la tecnología. Su objetivo se orienta a compaginar los procesos de las pruebas y los de organización, y hace una comparación de algunos modelos de madurez, pero no profundiza más allá de seleccionar la estructura para comparar procedimientos, es decir, no presenta un acercamiento al nivel de madurez de los modelos analizados ni tiene en cuenta la automatización como área a la que se puede evaluar su madurez.

García et al. [20] afirman que la calidad de los productos software está fuertemente influenciada por la calidad del proceso que los genera, y que particularmente la prueba contribuye en mayor medida. En este contexto su estudio pretende encontrar qué modelos de procesos de prueba han

sido definidos, adaptados o extendidos en la industria. Identificaron 23 modelos, muchos de ellos adaptados o extendidos desde Test Maturity Model Integration TMMi y TPI con diferentes arquitecturas, y desde la norma ISO/IEC/IEEE 29119 como enfoque arquitectónico alineado con otros modelos. Debido a que las métricas e indicadores utilizados por estos investigadores se orientaron a determinar para cada modelo el grado de adopción, la arquitectura, el domino, las fuentes utilizadas, las tendencias y la información mínima necesaria para comprenderlo, los resultados no determinan el nivel de madurez de la automatización de las pruebas en la industria y la investigación.

En el trabajo de Furtado et al. [4] se lee que la práctica de las pruebas de software es una de las maneras de producir productos de calidad, y que la automatización puede ser vista como una solución para probar la mayor cantidad de software en un tiempo determinado. Por eso su objetivo fue proponer directrices utilizando un modelo de madurez para la automatización, con el fin de ayudarlas a entrar gradualmente en esta práctica. Pero también afirman que la automatización puede no ser la solución para las necesidades de todas las empresas, por lo que su introducción puede complicar más de lo necesario el proceso de pruebas. Aunque esta investigación presenta una revisión de la literatura acerca de la automatización de las pruebas, su objetivo no es el de mostrar su nivel de madurez, sino justificar la propuesta de otro modelo para implementarla. Es decir, no analiza la madurez de la automatización como un tema de investigación y desarrollo de la industria del software.

3. MÉTODO

Se realizó una revisión sistemática de la literatura para encontrar el nivel de madurez actual de la automatización de las pruebas del software, siguiendo los procedimientos descritos por Serna [21]. La pregunta de investigación a responder fue: ¿Cuál es el nivel de madurez de la automatización de las pruebas como área de investigación y de desarrollo en la industria del software?

Para responder la pregunta se analizaron los trabajos que cumplieran con: cubrir expresamente a la automatización como área de investigación y desarrollo; describir modelos de madurez; presentar análisis comparativos a la eficiencia, la eficacia, el nivel de aceptación y la proyección; y realizar observaciones a la madurez de la automatización, los modelos o los procesos de prueba del software. Los trabajos seleccionados se limitaron a artículos publicados en revistas o actas de congresos, y reportes técnicos.

Se excluyó cualquier publicación que no describiera explícitamente un modelo de madurez relacionado con la automatización de las pruebas, no hiciera referencia a una matriz de comparación, no analizara el nivel de madurez del proceso de automatización, o no lo concibiera desde las perspectivas de investigación y desarrollo. La calidad de los aportes se validó con los procedimientos establecidos por el medio de publicación, es decir, la revisión por pares de las revistas y congresos como criterio principal. Se utilizaron las bases de datos IEEEXplore, la biblioteca digital ACM, Springer, Web of Science, ScienceDirect y Wiley Interscience.

Las razones para seleccionar están bases de datos es que concentran la mayor cantidad de revistas relacionados con la automatización de las pruebas del software, y a que recopilan las memorias de conferencias, simposios y publicaciones de organizaciones, normas y libros. La información es actualizada diariamente y permiten realizar búsquedas por múltiples opciones. Los términos de búsqueda de aplicaron en español e inglés.

4. RESULTADOS

Se recuperaron 978 trabajos. En una primera etapa se revisaron títulos y resúmenes, y se descartaron los irrelevantes y/o duplicados; posteriormente se analizaron los contenidos y se tuvo en cuenta solo a los describían completamente un modelo o analizaban la madurez de la automatización como área de investigación y desarrollo. Este proceso arrojó 16 modelos, que se describen en la Tabla 1, y 10 reportes de análisis a la eficiencia y eficacia de los modelos y a la madurez de la automatización. Para determinar la derivación subsecuente y para identificar los modelos originales, se clasificaron por año; de una u otra forma todos tienen en cuenta la automatización, aunque algunos presentan una orientación más marcada que otros.

Tabla 1. Modelos de madurez para la automatización de las pruebas del software

Modelo	Año	Orientación	Niveles	Escuela de pruebas [17]
MMAST [22]	1994	Automatización	4	Factory school
TAP [5, 23]	1995	Evaluación	5	Test-driven school
TCMM [24]	1996	Capacidad	4	Quality School
TSM [25]	1996	Capacidad	3	Quality School
TMM [26, 27]	1996	Procesos	5	Context-drive school
TIM [28]	1998	Mejoramiento	5	Standard School
TOM [29]	1998	Mejoramiento	3	Standard School
TPI [30]	1999	Mejoramiento	3	Standard School
ATLM [31]	1999	Automatización	5	Factory school
MB-V^2M^2 [32]	2002	Verificación y Validación	5	Control school
CB-VVCM [33]	2005	Verificación y Validación	5	Control school
SAMM [34]	2009	Aseguramiento	4	Context-drive school
CMMI-DEV [35]	2010	Calidad	4	Quality School
TMMi [36]	2012	Actividades de prueba y desarrollo	5	Analytical school
MPT.BR [37]	2012	Mejores prácticas	5	Agile School
ISO/IEC/IEEE [38]	2013	Estándares	6	Control school

En la Tabla 2 se aprecia los resultados del análisis a las opiniones publicadas acerca de la eficiencia y eficacia de los modelos de madurez de la automatización de las pruebas del software.

Tabla 2. Eficacia y eficiencia de la automatización de las pruebas

Modelo	Eficacia y eficiencia	Nivel de aceptación	Proyección
MMAST	Baja	Bajo	Ninguna
TAP	Baja	Medio	Poca
TCMM	Baja	Bajo	Ninguna
TSM	Baja	Bajo	Ninguna
TMM	Baja	Bajo	Ninguna
TIM	Baja	Bajo	Ninguna
TOM	Baja	Bajo	Ninguna
TPI	Media	Alta	Alta
ATLM	Baja	Bajo	Poca
MB-V^2M^2	Media	Bajo	Poca
CB-VVCM	Baja	Bajo	Poca
SAMM	Media	Medio	Alta
CMMI-DEV	Alta	Alto	Alto
TMMi	Alta	Alto	Alta
MPT.BR	Alta (Brasil)	Alto (Brasil)	Alta (Brasil)
ISO/IEC/IEEE	Media	Medio	Alta

Esta valoración de los modelos se resume de los trabajos recolectados en los que: 1) se describe el modelo o se analiza su aplicación en casos de la industria [10, 12, 15, 39-41]; 2) el nivel de

aceptación es la ponderación a la valoración que los autores hacen de cada uno [16, 41, 43, 44]; y 3) la proyección demuestra si el modelo tiene una vida útil referente o si fue efímero su paso por la industria [10, 16, 39, 41, 43].

5. MODELO DE MADUREZ PROPUESTO

Conocer el potencial de la automatización de las pruebas del software será posible en la medida en que las organizaciones aprovechen sus beneficios, y esto se logra ubicando el proceso mismo dentro de un nivel de madurez. Cuanto más maduro sea el proceso mayor será la eficiencia y la eficacia del plan de pruebas. Aunque la mayoría de investigadores y profesionales están familiarizados con los niveles de madurez utilizados por Capability Maturity Model CMM, la razón de proponer un modelo con otra escala de valoración es que CMM se centra en evaluar la evolución de los procesos de una organización, y en este trabajo los autores quieren acercarse al estado de la automatización como área de investigación y desarrollo. Es decir, la idea no es presentar la evolución, sino el estado en el que se encuentra al momento de la revisión.

El objetivo de este modelo es determinar la madurez de la automatización de las pruebas del software, analizándola desde una perspectiva y con niveles comparativos a la madurez de los seres humanos, es decir: 0. Infantil, 1. Adolescente, 2. Adulto y 3. Veterano.

- Nivel 0: *Infantil.* El proceso de automatización de las pruebas está en un nivel de madurez *Infantil* cuando necesita mucho cuidado, atención y afecto. Las características del nivel son:

  - La mayoría de las pruebas se ejecuta al finalizar el desarrollo del producto, porque el plan de pruebas automatizado no está en sintonía con el ciclo de vida del desarrollo.
  - La cantidad de errores detectados hace que la prueba falle y que probablemente se detenga la secuencia de comandos automatizados, porque las incoherencias entre el software bajo prueba y el marco de automatización invalida cualquier caso de prueba que se aplica.
  - Se genera procesos de reingeniería, porque en cualquier caso hay que corregir el error o actualizar el caso de prueba, y luego volver a ejecutar el plan de pruebas para encontrar el siguiente problema.
  - El equipo invierte más tiempo trabajando en los casos de prueba que en su automatización.
  - La mayoría de organizaciones que intenta introducir la automatización de las pruebas no tarda en retroceder al proceso manual. Lamentablemente se dan cuenta tarde de la madurez de la automatización, porque normalmente les demora entre dos y diez veces más tiempo que el proceso manual.
  - Debido a que hay que cuidar, mimar y prestarle mucha atención, la automatización no se puede dejar sin vigilancia por mucho tiempo. Aquí es posible afirmar que la madurez de esta tecnología desalienta en lugar de alentar su introducción.

- Nivel 1: *Adolescente*. El proceso de automatización se encuentra en el nivel *Adolescente* cuando es posible aplicar el plan de pruebas, y el conjunto de casos de prueba, y dejarlo solo y sin vigilancia durante un tiempo razonable, tal vez un par de horas o incluso una noche, pero todavía se desconfía de su responsabilidad. Las características de este nivel son:

  - Es importante conocer el error, pero no se necesitará decenas de casos de prueba para demostrarlo.
  - Un solo error en el software bajo prueba hace que falle gran cantidad de casos de prueba.

- Se desperdicia mucho tiempo intentando analizar la causa de cada bloqueo a la vez que se deja de ejecutar muchas pruebas.
- Para encontrar las fallas el equipo debe solucionar los problemas y volver a correr la automatización, un ciclo que se tiene que repetir muchas veces.
- En este nivel y al igual que con los adolescentes, el proceso de automatización es sorprendentemente útil, pero se comporta de manera irresponsable y puede causar más daño que beneficio.

- Nivel 2: *Adulto*. El proceso de pruebas automatizadas se encuentra en el nivel *Adulto* cuando es digno de confianza y se puede dejar solo para que funcione sin vigilancia durante largo tiempo, por ejemplo, durante todo un fin de semana, pero aun así todavía se desvía de sus responsabilidades. Las características de este nivel son:

  - Al final el proceso de prueba arroja mucha información útil.
  - Aunque quizás la mayoría de casos de prueba ha fallado, los datos son diferentes y sobre todo reportan algo del software bajo prueba.
  - La automatización es algo más que secuencias de comandos.
  - El equipo se concentra en encontrar y corregir los problemas reportados, mientras los casos de prueba se continúan ejecutado sin intervención.
  - En este nivel la automatización no requiere vigilancia extrema y, aunque el proceso es responsable y se puede confiar en él, todavía no es auto-dirigido, porque su madurez no ha llegado a ese nivel de independencia.

- Nivel 3: *Veterano*. Para llegar al nivel de madurez *Veterano* la automatización de las pruebas del software tiene que haber recorrido y crecido a través de los anteriores; en este momento es posible dejarlo para que la naturaleza siga su curso. En este nivel la automatización es totalmente gestionable, las herramientas disponibles son autónomas, los casos de prueba son repetibles y el proceso se puede dejar funcionando sin vigilancia por semanas. Las características de este nivel son:

  - El plan de pruebas y los casos de prueba son eficientes y eficaces, y el equipo observa todo el proceso de automatización como una disciplina no como un arte.
  - El proceso de la automatización utiliza la reutilización como base porque ahora es bien entendido y aplicado.
  - Al final se tiene un conjunto de casos de prueba validado y maduro, y una serie de reportes que denotan la calidad y fiabilidad del producto y de la automatización de las pruebas.
  - El plan de pruebas se estructura como un enfoque planificado que involucra el diseño de los casos de prueba y el mantenimiento del plan de pruebas.
  - Es posible aplicar procedimientos de gestión y de planeación estratégica a todos los aspectos de la automatización.
  - La automatización de las pruebas es un proceso autónomo en cuanto a reproducción y selección de los valores de entrada, y a la validación y exposición de resultados.
  - La organización cuenta con un banco de pruebas automatizadas reutilizable y fácil de proyectar a cada producto bajo prueba.
  - Como en el caso de los humanos, en este nivel la automatización aporta experiencia y madurez, y las pone al servicio de los demás procesos del software.

5.1 Madurez de la automatización de las pruebas del software

Con base en los resultados presentados en las Tablas 1 y 2, y en las opiniones, reflexiones y críticas que la industria y los investigadores manifiestan acerca de los modelos de madurez y de la automatización misma, en la Tabla 3 se resume los resultados acerca de la madurez de este proceso. En la primera columna se ubica las etapas tradicionales de un proceso de pruebas de software, y en las demás los niveles del modelo de madurez de la automatización de las pruebas del software propuesto en esta investigación.

Tabla 3. Madurez del desarrollo de la automatización de las pruebas

Etapas del proceso	Nivel de madurez			
	Infantil	Adolescente	Adulto	Veterano
Percepción				
Sensibilización				
Decisión				
Implementación				
Apropiación				
Consolidación				
Institucionalización				
Externalización				
Nivel de madurez actual de la automatización de las pruebas				

6. ANÁLISIS DE RESULTADOS

Es importante observar que se ha incrementado el interés de los investigadores y la industria por proponer modelos para automatizar el proceso de las pruebas del software, en parte motivados por apoyar el mejoramiento de la calidad y la escalabilidad de sus productos. Pero también llama la atención que, aunque se presenta diversos modelos de madurez, a la industria parece que le faltara algo en este tema, y es que no los aplica rigurosamente. Una de las principales cosas que se aprende con estos modelos es que, cuando se avanza a lo largo de ellos, es muy importante no saltarse los niveles. Ese es el punto para que sea un modelo de madurez. Como se propone en este trabajo las personas crecen a través de cada etapa y a partir de lo que ya son en cada una. No es posible dejar de ser niño para pasar automáticamente a ser adulto, aunque se intente desesperadamente, porque estará destinado al fracaso si intenta saltar la adolescencia.

Al analizar la aplicación de un modelo de automatización de pruebas y constatar que la empresa se encuentra en un nivel alto, se podría argumentar que simplemente es un caso en el que pasó de la implementación manual, y aprendió en los demás niveles, hasta lograr el despliegue total de la automatización. Pero la realidad en los resultados de esta investigación es que pocas empresas tienen la paciencia y los recursos para atender el proceso completo a través de todos los niveles, hasta lograr una automatización madura. La mayoría que califica los modelos como deficientes o ineficaces es porque han intentado saltarse niveles para avanzar.

Por otro lado, por años la industria del software ha tratado y ha invertido en mejorar la calidad de sus productos, pero ha sido una tarea difícil, porque los clientes se han vuelto cada vez más exigentes y los sistemas software más complejos. El aseguramiento de la calidad se está convirtiendo en una condición permanente para la sobrevivencia de las empresas, y actualmente es una realidad en la industria del software. A pesar de que algunos investigadores [48-50] afirman que la calidad del software ha mejorado en las últimas décadas, todavía está muy lejos del escenario ideal y de la madurez esperada.

Para llenar este vacío la industria ha tratado de adaptar diferentes modelos de madurez a sus procesos del ciclo de vida, pero de acuerdo con los reportes analizados en esta investigación esos modelos describen prácticas de Verificación y Validación limitadas, que no se centran directamente en la madurez del proceso de automatización de las pruebas. Esta madurez se puede definir como una forma de medir el nivel de capacidad que tiene una organización para gestionar los planes de pruebas de los proyectos [48], y el principal objetivo de conocerla es ayudarle a mejorar la construcción del software.

Aunque la prueba es un elemento esencial para lograr la satisfacción del cliente y una parte integral de todo el ciclo de vida del desarrollo de software, requiere velocidad, eficiencia y flexibilidad. Por eso es que el papel de las pruebas automatizadas es el de apoyarlo para eliminar tareas mecánicas, rutinarias y lentas. Debido a esta exigencia en velocidad, la gestión de los datos de prueba, y de los diferentes entornos de plan de pruebas, necesita ser muy eficiente y eficaz. Esta característica incrementa continuamente la demanda por una automatización madura, que asegure que el proceso de pruebas ocurre sobre una base muy regular [49]. Además, que facilite la construcción de escenarios cada vez más predecibles, que requieran menos esfuerzo y que les permita a los equipos de desarrollo y de prueba obtener información instantánea sobre la calidad del sistema en producción. De acuerdo con los resultados analizados en esta investigación esas características no se logran.

La prueba automatizada es una estrategia fiable y, para muchas empresas, es la única opción para optimizar los estándares de calidad del software [50]. Pero de acuerdo con los resultados de esta investigación todavía se encuentra dentro de los tiempos de maduración que requiere cualquier aplicación compleja. Por otro lado, en la práctica los diferentes segmentos de aplicación de la automatización se encuentran en diferentes estados de madurez, porque en cada etapa aparece un aprendizaje único que brinda la oportunidad de tener el poder para pasar a la etapa superior. Uno de los problemas detectados es que actualmente en la industria la automatización de las pruebas se gestiona como otro proceso del desarrollo de software, y las herramientas disponibles para aplicarla se ofrecen de acuerdo con la demanda del mercado, no por una planeación real de las necesidades de la industria de un sistema escalable y mantenible para lograrlo. Esto genera algunos desafíos que es necesario afrontar para lograr que la automatización de las pruebas logre realmente su objetivo de alcanzar el nivel de madurez *Veterano*:

- El tiempo que se invierte para automatizar es extenso, porque la industria no tiene un adecuado nivel de madurez de sus propios procesos de desarrollo.
- El mantenimiento del plan de pruebas, y del conjunto de casos de prueba, es muy frecuente y no se adapta fácilmente a los escenarios cambiantes de los sistemas.
- Para los miles de líneas de código de la automatización no hay documentación, y la que existe no es adecuada o está desactualizada.
- La industria tiene el problema de que toda la vida ha aplicado pruebas manuales y ha invertido bastante en capacitar a sus equipos de probadores. Pero resulta que no saben automatizar el plan de pruebas, simplemente porque no saben de programación.
- La industria actual invierte dinero, tiempo y esfuerzo solamente para hacer su trabajo, es decir, desarrollar, pero al automatizar las pruebas se encuentra con la frustración de no obtener rápidamente resultados.
- Las habilidades para automatizar las pruebas no son fáciles de adquirir, porque cada herramienta tiene un enfoque único y patentado para interactuar con las tecnologías de desarrollo. Por lo que la industria necesita capacitar diferentes equipos de prueba para atender

las demandas de cada sistema que desarrolla, y debe utilizar diferentes herramientas debido a los diversos requerimientos de los clientes en cuanto a lenguajes de programación.

- La rotación de recursos crea caos que cambia el enfoque completo del plan de pruebas, por lo que su automatización no está lista cuando se necesita.
- El otro desafío importante para que la automatización de las pruebas madure como área de investigación y de desarrollo, es el costo de las herramientas y de la adquisición de las habilidades especiales para aplicarlas. Gran parte de la industria opta por la prueba manual, porque a veces su automatización es más costosa que el mismo desarrollo del sistema.

De acuerdo a los resultados de esta investigación muchas empresas dedicadas al desarrollo de software ven y aplican la automatización de las pruebas como una especie de *bala de plata*, que resolverá todos sus problemas de calidad, les ayudará a cumplir con todos los requisitos de los sistemas y les ahorrará mucho tiempo y esfuerzo. Este no es el caso, porque esta implementación implica una curva de aprendizaje progresivo hasta alcanzar la madurez adecuada. Eso significa que durante los primeros niveles las pruebas automatizadas pueden aumentar el esfuerzo y el costo de aplicación. Tal como sucede con la generación del plan de pruebas, un área clave donde las expectativas de la automatización parecen estancadas en el tiempo, porque actualmente se debe hacer de forma manual, aunque en el mercado haya herramientas que la apoyan, pero ninguna es automática. También es importante destacar que todavía no se encuentra una herramienta que apoye todos los entornos de sistemas operativos y lenguajes de programación. Esto significa que la mayoría requiere un conjunto de herramientas diversas para automatizar sus pruebas, porque no hay tal cosa como una de *talla única*.

Por otra parte, las empresas deciden enfrascarse en la automatización porque piensan que la reducción de costos y de tiempos de entrega les permitirá cumplir con los plazos. Pero actualmente es poco probable que la automatización reduzca inmediatamente el esfuerzo de la prueba y ahorre el tiempo necesario, debido a que se requiere capacitación del personal para utilizar las herramientas y para aprender las diversas maneras eficaces de hacerlo. Además, la escritura de *scripts* de prueba aporta un nuevo nivel de complejidad al plan de pruebas, lo que requiere que los probadores y las empresas piensen en términos de fiabilidad y reutilización en el diseño, en lugar de simplemente en la eficacia de la prueba.

Otro asunto que la industria no comprende a cabalidad es que no todo el plan de pruebas se puede automatizar. Muchos sistemas contienen controles de terceros o *widgets* para mejorar su funcionalidad, lo que representa un problema, porque es poco probable que una herramienta de automatización verifique su compatibilidad con todos estos controles, y puede que no sea capaz de manipular los caminos necesarios para probar la aplicación. Además, pruebas como la comprobación de que un documento se imprime correctamente no se pueden automatizar, o tampoco es rentable para aquellos casos de prueba que solamente se ejecutan una vez.

Algo un poco más alejado de la realidad es que a menudo se espera que la automatización permitirá realizar pruebas al ciento por ciento de la aplicación. La industria debe comprender que la prueba es una tarea potencialmente infinita, sin embargo, si se centra en las áreas clave del código puede mejorar considerablemente la fiabilidad del software. En términos generales, hasta el momento la automatización de las pruebas del software solamente permite: 1) incrementar la fiabilidad del sistema, 2) mejorar la calidad de las pruebas, 3) disminuir el esfuerzo que se dedica a las pruebas, y 4) reducir el cronograma del proyecto. Por todo esto es posible concluir que actualmente el nivel de madurez de la automatización de las pruebas del software como área de investigación y desarrollo es *Adolescente.*

7. CONCLUSIONES

Una lección que se puede aprender en esta investigación es que, en lo referente a la madurez de la automatización de las pruebas del software, el vaso no está vacío, pero tampoco está lleno, solamente está medio vacío. Esta apreciación se sustenta en los resultados analizados y a que se percibe una conciencia cada vez mayor, de quienes tienen experiencia en este campo, de que muchos esfuerzos en la automatización de las pruebas no cumplen las expectativas. Para llegar a esta conclusión la investigación encontró que, aunque existe mucho esfuerzo orientado al desarrollo y el mantenimiento de la automatización, es muy importante realizar un análisis costo-beneficio a cualquier intento por implementarla. Esto significa analizar los resultados y las experiencias en diferentes empresas y métodos, porque los éxitos reportados en la literatura han sido mayormente en áreas en las que tenía sentido automatizar algunas pruebas, en lugar de todo el plan. Además, al equipo lo asesoran especialistas y se les permite el tiempo para hacerlo bien. Esto no es un denominador común a toda la industria y para todos los sistemas.

Aunque la automatización puede añadir complejidad y costos al esfuerzo de un equipo de pruebas, también puede proporcionar cierta ayuda valiosa si se cuenta con las personas y el entorno adecuados, y si se aplica cuando tiene sentido hacerlo [11].

Es importante definir el propósito de iniciar un esfuerzo de automatización de las pruebas porque, aunque existe varias categorías de herramientas para hacerlo cada una tiene su propio propósito.

Identificar qué se desea automatizar y en qué fase del ciclo de vida implementarlo es el primer paso para desarrollar una estrategia de automatización. Solamente desear que todo sea probado más rápido no es una estrategia práctica, hay que ser específico.

Desarrollar una estrategia de automatización es más importante que decidir qué se va a automatizar, cómo se va a hacer, cómo se mantendrán los *scripts*, y cuáles serán los costos y los beneficios que se espera. Al igual que con todos los esfuerzos de prueba se debe construir una estrategia o plan de pruebas para implementarla.

Muchas herramientas de prueba son sofisticadas y utilizan lenguajes existentes o código propietario, por lo que el esfuerzo de la automatización se convierte en una rutina manual que no es diferente del trabajo de un programador, codificando en un determinado lenguaje para escribir programas para automatizar un proceso de prueba. Hay que tratar a todo el proceso de la automatización como si fuera un esfuerzo de Ingeniería del Software, es decir, definir lo que se automatizará (requisitos); diseñar la automatización; escribir, probar e implementar los *scripts*; hacerle mantenimiento y definir su vida útil.

Para alcanzar sus beneficios hay que observar el esfuerzo de la automatización como una inversión que requiere tiempo y recursos. Esto implica que por lo general las primeras versiones del sistema no ofrecen la recompensa esperada, y que el beneficio viene luego de correr las pruebas automatizadas en cada lanzamiento posterior. De ahí la importancia de ubicar rápidamente la automatización de las pruebas en alguno de los niveles de madurez propuestos en esta investigación.

Debido a que la automatización es realmente otro esfuerzo de desarrollo de software, es importante que quienes realizan el trabajo posean las habilidades y destrezas necesarias. Un buen probador no significa necesariamente un buen automatizador. Los buenos probadores todavía serán necesarios para identificar y escribir casos de prueba, pero se necesita al automatizador

para que tome esos casos de prueba y escriba código para su automatización. Esto no quiere decir que los probadores no puedan aprender a ser automatizadores, es solo que esos dos roles son diferentes y las habilidades necesarias también son diferentes.

Los modelos analizados en este trabajo han sufrido cambios en el tiempo, proporcionándole a la industria nuevas versiones y en periodos cada vez más cortos. Esto dificulta su apropiación y experimentación, a la vez que la pérdida de respaldo por parte de investigadores y practicantes.

Al evaluar el proceso de automatización de las pruebas a partir de las etapas del modelo propuesto, y debido a que las etapas de apropiación, consolidación, institucionalización y externalización se encuentran en nivel *Infantil*, que las etapas de decisión e implementación en nivel *adolecente*, y las de percepción y sensibilidad en nivel *Adulto*, los autores concluyen que la madurez de la automatización, como área de investigación y desarrollo se encuentra en un nivel *Adolecente*.

De acuerdo al análisis a los resultados publicados por los investigadores y la industria, y a la aplicación del modelo de madurez propuesto, la automatización de las pruebas del software se encuentra actualmente en desarrollo y en proceso de maduración, por eso no es de esperar que en corto tiempo cumpla las promesas que viene haciendo desde hace décadas. El trabajo para alcanzarlas debe continuar y cada vez se deberá sumar más investigadores y empresas para lograrlo.

El futuro de la automatización es prometedor, pero intentar proyectar lo que logrará más adelante es una lotería, y ese objetivo está por fuera de esta investigación. Lo que sí se puede afirmar, con base en los resultados encontrados y descritos aquí, es que actualmente la madurez de la automatización de las pruebas del software está en un amplio proceso de aprendizaje y de experimentación, y que como sucede con los humanos adolescentes: *se puede confiar en ella, pero no dejarla sola mucho tiempo*.

REFERENCIAS

[1] Serna E. (2012). Social control for science and technology. En Tenth Latin American and Caribbean Conference for Engineering and Technology. Panamá, Panamá.

[2] Myers J. (1979). The art of software testing. Wiley.

[3] Heiskanen H. et al. (2012). A Test Process Improvement Model for Automated Test Generation. En O. Dieste et al. (Eds.), PROFES 2012. Springer.

[4] Furtado A. et al. (2014). Towards a maturity model in software testing automation. En Ninth International Conference on Software Engineering Advances. New York, USA.

[5] Paulk M. et al. (1993). Key practices of the capability maturity model. Technical report CMU/SEI-93-TR-25. Carnegie Mellon University.

[6] ISO/IEEE. (2013). ISO/IEEE 29119 – Part I International Standard. Software and systems engineering/software testing, concepts and definitions. IEEE.

[7] Hass A. (2008). A guide to advanced software testing. Artech House.

[8] Serna E. (2013). Functional test of software - A Constant Verification process. Fondo Editorial ITM.

[9] Serna M. y Arango F. (2010). Effectiveness analysis of the set of test cases generated with the Requirements by Contracts technique. En V Congreso Colombiano de Computación. Cartagena, Colombia.

[10] Karthikeya S. y Rao S. (2014). Adopting the right software test maturity assessment model. En Cognizant 20-20 Insights. New Jersey, USA.

[11] Anya P. y Smith G. (2014). Qualitative research methods in Software Engineering. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 4(2), 14-18.

[12] Kumar P. (2012). Test process improvement – Evaluation of available models. Maveric's Point of View.

[13] ISTQB. (2012). Standard glossary of terms used in software testing. International Software Testing Qualifications Board.
[14] Grottke M. (1999). Software Process Maturity Model study. IST-1999-55017. PETS Project.
[15] Swinkels R. (2000). A comparison of TMM and other Test Process Improvement Models. 12-4-1-FP Report. Technische Universiteit Eindhoven.
[16] Kulkarni S. (2006). Test process maturity models - Yesterday, today and tomorrow. En 6th Annual International Software Testing Conference. Delhi, India.
[17] de Souza G. (2007). Modelo de maturidade em testes com foco em ambientes de testes hetorogêneos. Mestrado tesis. Universidade Federal de Pernambuco.
[18] Sulayman M. (2009). A systematic literature review of software process improvement for small and medium web companies. Communications in Computer and Information Science 59, 1-8.
[19] von Wangenheim C. et al. (2010). Systematic literature review of software process capability/maturity models. En International Conference on Software Process Improvement and Capability Determination. Pisa, Italy.
[20] García C. et al. (2014). Test process models: Systematic literature review. Communications in Computer and Information Science 477, 84-93.
[21] Serna E. (2018). Metodología de investigación aplicada. En E. Serna (Ed.), Ingeniería: Realidad de una Disciplina (pp. 6-33). Editorial Instituto Antioqueño de Investigación.
[22] Krause M. (1994). Maturity model for automated software testing. Medical Device & Diagnostic Industry Magazine (December), 1-10.
[23] Paulk M. et al. (1993). Capability Maturity Model. IEEE Software 10(4), 18-27.
[24] Burgess S. y Drabick R. (1996). The I.T.B.G. Testing Capability Maturity Model. Recuperado: http://www.iti-inc.com/TCMM.htm
[25] Gelperin D. (1996). A Testability Support Model. En Fifth International Conference on Software Testing, Analysis & Review. Orlando, USA.
[26] Burnstein I. et al. (1996). The Development of a Testing Maturity Model. En Ninth International Quality Week Conference. San Francisco, USA.
[27] Burnstein I. et al. (1996). Developing a Testing Maturity Model. CROSSTALK, Software Technology Support Center.
[28] Ericson T. et al. (1998). TIM - A Test Improvement Model. Software Testing Verification and Reliability 7(4), 229-246.
[29] Systeme Evolutif. (1998). Test Organization Maturity Model. Recuperado: https://gerrardconsulting.com/mainsite/?q=node%2F485
[30] Koomen T. y Pol M. (1999). Test process improvement: A practical step-by-step guide to structured testing. Addison-Wesley.
[31] Dustin E. et al. (1999). Automated Software Testing - Introduction, management, and performance. Addison-Wesley.
[32] Jacobs J. y Trienekens J. (2002). Towards a metrics based Verification and Validation maturity model. En 10th International Workshop on Software Technology and Engineering Practice. Montréal, Canada.
[33] Yoon K. et al. (2005). A framework for the V&V capability assessment focused on the safety-criticality. En 13th IEEE International Workshop on Software Technology and Engineering Practice. Budapest, Hungary.
[34] Chandra P. (2009). Software assurance maturity model - A guide to building security into software development, Version - 1.0. Open Web Application Security Project.
[35] CMMI-DEV. (2010). CMMI for Development, Version 1.3. CMU/SEI-2010-TR-033. Software Engineering Institute.
[36] TMMI. (2012). Test Maturity Model Integration, Release 1.0. TMMi Foundation.
[37] Furtado A. et al. (2012). MPT.BR: A Brazilian maturity model for testing. En 12th International Conference on Quality Software. Xi'an, China.
[38] ISO/IEC. (2013). Software and systems engineering – Software testing, Part 2: Test processes. ISO/IEC/IEEE 29119-2:2013.
[39] Mosley D. y Posey B. (2002). Just enough software test automation. Prentice-Hall.
[40] Meszaros G. (2007). xUnit test patterns: Refactoring test code. Addison-Wesley.
[41] Bhaggan K. (2009). Test automation in practice. Master's Thesis. DSW Zorgverzekeraar.

[42] Balaraman R. y Krishnankutty H. (2013). Need for a comprehensive test maturity model. En Infosys. Bangalore, India.
[43] Kohlegger M. et al. (2009). Understanding maturity models results of a structured content analysis. En I-KNOW ’09 and I-SEMANTICS ’09. Graz, Austria.
[44] Wiklund K. et al. (2012). Technical debt in test automation. En Fifth International Conference on Software Testing, Verification and Validation. Montreal, Canada.
[45] Prado D. (2003). Project Management in Organizations. Editora de Desenvolvimento Gerencial.
[46] Lionbridge. (2009). Test process assessments move into the real world - A Rethinking QA white paper. Lionbridge Technologies.
[47] Alsmadi I. (2012). Advanced automated software testing: Frameworks for refined practice. En Information Science Reference. New York, USA.
[48] Nasir N. y Sahibuddin S. (2011). Critical success factors for software projects: A comparative study. Scientific Research and Essays 6(10), 2174-2186.
[49] SGI. (2010). Modernization - Clearing a pathway to success. The Standish Group International Inc.
[50] Cangussu J. et al. (2002). A formal model of the software test process. IEEE Transactions on Software Engineering 28(8), 782-796.

CAPÍTULO XIX

Integración de propiedades de la realidad virtual, las redes neuronales artificiales y la inteligencia artificial en la automatización de las pruebas del software: Una revisión[1]

Edgar Serna M.[1]
Eder Acevedo M.[2]
Alexei Serna A.[1]
[1]Instituto Antioqueño de Investigación
[2]Universidad Cooperativa de Colombia

La automatización completa de las pruebas del software se considera, hasta el momento, como una meta poco factible de alcanzar. En este capítulo se concluye que, con los últimos descubrimientos e innovaciones en las áreas de la Realidad Virtual VR, las Redes Neuronales Artificiales ANN y la Inteligencia Artificial AI, se abre una amplia posibilidad de lograrlo o, por lo menos, de acercarse mucho a esta meta. En este trabajo se describe una propuesta teórica para integrar propiedades de cada una de ellas en un proceso de automatización de las pruebas del software. Se parte de una clasificación y descripción de las mismas y, luego de consultar la literatura, de entrevistar y dialogar con especialistas de Australia, Estados Unidos, Alemania y Colombia, y de adicionar la experiencia de los investigadores, se propone la construcción de dos herramientas: 1) un robot para diseñar y aplicar pruebas funcionales, y 2) una máquina virtual para encontrar los errores en la estructura lógica del código (pruebas estructurales). Se espera que con ambas herramientas se reemplace el factor humano, con las ventajas que esto significa, de tal manera que la primera encuentre fallas de procedimiento y la segunda errores de funcionamiento.

[1] Publicado en inglés en Journal Software: Evolution and Process 31(7), e2159. 2019.

INTRODUCCIÓN

Desde hace décadas uno de los objetivos de la investigación en Ciencias Computacionales es lograr automatizar la prueba del software, pero ¿por qué? La realidad es que somos una sociedad software-dependiente [1] y, en la realidad actual del desarrollo tecnológico y del creciente consumo de este producto, se ha incrementado la demanda por el mejoramiento a la calidad del software; además, el ritmo al que se debe producir también tiende a incrementarse. Esta situación ha hecho que la industria tenga que replantear sus metodologías de trabajo para responder las demandas de la sociedad y del mercado, satisfacer las necesidades de los clientes y mantener la oferta cubierta y actualizada, aunque espera lograrlo sin invertir demasiado tiempo y dinero.

Por otro lado, el escenario se vuelve cada vez más competido ya que la globalización y el libre mercado ofrecen oportunidades de adquirir software en cualquier parte del mundo. En todo caso, la industria debe verificar y validar cada sistema para evaluar su calidad, un proceso en el que se diseña un conjunto de casos de prueba que, en la mayoría de casos, se aplica manualmente y conlleva funciones repetitivas que exigen tiempo [2].

Por todo esto se recurre a la automatización, con el objetivo de acelerar procesos y disminuir costos, además, porque la prueba puede proporcionar mayor cobertura. Otra de sus ventajas es que se puede ejecutar sin supervisión, con muy poca intervención humana y en horarios contrarios a los laborales. De esta manera se incrementa la productividad en el área de control de calidad, a la vez que el producto se puede entregar en menos tiempo y con ahorro de costos.

Pero, a pesar de llevar décadas de investigación, experimentación y aplicación, todavía no se puede afirmar que las pruebas del software estén totalmente automatizadas. Si bien se ha logrado progresos y en el mercado se consigue herramientas poderosas para este propósito, aun no se avanza en la misma medida que la sociedad exige calidad, fiabilidad y seguridad en los productos. Por su parte la industria les atribuye esta situación a los costos de la automatización, a la falta de personal calificado y a la ambigüedad de los documentos de especificación de requisitos, porque generan retrasos en el desarrollo y, al final, no tiene tiempo suficiente para ejecutar las pruebas funcionales y estructurales establecidas para el sistema.

En este sentido algunos científicos computacionales [1, 3] están desarrollando ideas innovadoras que se podrían integrar a un proceso de automatización de las pruebas. Entre otras, integran propiedades de la Realidad Virtual, las Redes Neuronales Artificiales y la Inteligencia Artificial, con resultados de investigación que demuestran progreso y casos de éxito. Uno de los inconvenientes que presentan estas investigaciones es que no trabajan de manera transdisciplinar, es decir, no experimentan, o lo hacen casi imperceptiblemente, combinando a la vez las propiedades de varias de esas áreas. Por eso el objetivo de este trabajo es analizar, desde una perspectiva teórica, si es posible integrar, ajustar y armonizar sus propiedades en un proceso para automatizar las pruebas del software.

La investigación desde la que se origina este trabajo se desarrolla en dos fases: 1) orientada a encontrar propiedades, principios y características representativas de VR, ANN y AI, y 2) mediante una amplia revisión sistemática de la literatura, encuestas y diálogos con especialistas de Australia, Estados Unidos, Alemania y Colombia, además de los aportes desde el conocimiento y la experiencia de los investigadores, valorar su integración en la automatización de las pruebas del software y describir una propuesta teórica de cómo lograrlo. Los autores ya publicaron los resultados de la primera fase [4-6]. En este capítulo se divulga los resultados de la segunda fase.

1. MÉTODO

De acuerdo con la naturaleza, el tema y el alcance de la investigación el método aplicado es analítico-inferencial, pero también se utiliza puntos de vista y sugerencias de personas especializadas en cada una de las temáticas consultadas, además de en las pruebas del software, tanto manuales como automáticas; por lo que la investigación también se cataloga como de desarrollo. Para la recopilación de los datos y el análisis de pertinencia y representatividad de las propiedades objetivo, primero se realizó una revisión sistemática de la literatura acerca de las propiedades de la Realidad Virtual, las Redes Neuronales Artificiales y la Inteligencia Artificial.

Posteriormente, y mediante un trabajo conjunto entre los investigadores y los especialistas, se evaluaron, complementaron y direccionaron las propiedades hasta identificar y documentar las más pertinentes para el logro del objetivo de la investigación: la posibilidad de integrarlas en la automatización de las pruebas del software. La búsqueda se realizó en las bases de datos de ACM, IEEE, Sciendirect y Wos, mientras que la población de especialistas se conformó con representantes de la industria y la academia de Australia, Estados Unidos, Alemania y Colombia.

En el muestreo los investigadores analizaron los hallazgos en las entrevistas y puntos de vista de los especialistas, simultáneamente con los datos recolectados, además de su experiencia, hasta alcanzar la saturación de datos teóricos. En total se seleccionaron 42 trabajos y se dialogó con 21 especialistas en las áreas específicas, las pruebas del software y/o la automatización de las pruebas del software. Durante las dos primeras fases (revisión sistemática de la literatura y diálogos con especialistas), y mediante integración permanente, se construyó el listado de propiedades de cada área identificadas como factibles para integrarlas en la automatización de las pruebas del software. Posteriormente, y luego de analizar y discutir las propiedades documentadas, se eliminaron las que obtuvieron menor valoración y se conformó la muestra final.

Antes de integrarlas en la muestra se validaron y verificaron bajo la asesoría de los especialistas, y mediante una escala de valoración definida a partir de los resultados en los trabajos consultados, las opiniones y las recomendaciones de los especialistas, además de la experiencia de los investigadores. En consecuencia, se definió una escala de diferencial semántico de cinco grados acerca de la probabilidad de integrar cada una en la automatización: *Muy débil*, *Débil*, *Medio*, *Fuerte* y *Muy fuerte*. Para finalizar, y en respuesta al objetivo de la investigación, se hizo una triangulación de las valoraciones para determinar las de mayor probabilidad de integración transdisciplinar en la automatización de las pruebas.

2. RESULTADOS

2.1 Trabajos relacionados

Como se mencionó antes, la investigación acerca de cómo relacionar las propiedades de VR, ANN y AI en las pruebas del software se trabaja de forma independiente, es decir, tomando algunos conceptos de cada área individual e involucrándolos en las fases del ciclo de vida. La idea de la investigación que genera este trabajo es describir cómo interrelacionar, de forma transdisciplinar, propiedades de las tres áreas en un proceso de automatización, asumido como paralelo al ciclo de vida y con la visión de mejorar la calidad, fiabilidad y seguridad de los sistemas.

En las Tablas 1 a 3 se presenta los trabajos representativos, hallados en la revisión sistemática de la literatura, que presentan resultados disciplinares al integrar esas propiedades a las pruebas del software.

Tabla 1. Integración de propiedades de la Realidad Virtual en las pruebas del software

Ref.	Descripción
[7]	Plantea cómo incorporar características y capacidades de los ambientes virtuales en las pruebas del software, los gráficos, el hardware de soporte y las interfaces.
[8]	Describe una técnica para realizar la prueba mediante la interacción en aplicaciones VR y propone una arquitectura de pruebas para hacerlo.
[9]	Presenta una propuesta de pruebas utilizando simulación de VR para que el usuario revise los productos de forma virtual, para reducir los errores y contribuir a un mejor diseño.
[10]	Muestra algunas propiedades de los sistemas VR que podrían ser útiles en las pruebas virtuales, con la idea de superar los problemas de las manuales.
[11]	Aplica una propuesta de prueba basada en un sistema RV, encontrando mejora significativa en la calidad del producto.
[12]	Investiga acerca de la reutilización del software en entornos virtuales y concluye que, en las últimas dos décadas, se ha incrementado el desarrollo de nuevas herramientas.
[13]	Presenta una herramienta para utilizar VR en la realización de pruebas, que se podría adecuar para probar diferentes contextos.
[14]	Proporciona nuevos modos de interacción virtual, que se pueden utilizar como herramientas para automatizar las pruebas del software.

Tabla 2. Integración de propiedades de las Redes Neuronales Artificiales en las pruebas del software

Ref.	Descripción
[15]	Propone modelos ANN para identificar los módulos software del sistema más propensos a fallas, con el objetivo de identificar los que necesitan mayor atención en el desarrollo.
[16]	Utiliza propiedades de ANN para predecir el esfuerzo necesario para ejecutar las pruebas del software.
[17]	Describe un *oráculo* automatizado para respaldar las actividades de prueba manual y reducir costos y tiempos.
[18]	Utiliza propiedades de las ANN como *oráculo* de prueba automatizado.
[19]	Propone un método basado en propiedades de ANN para ayudarles a los desarrolladores a elegir entre métodos automáticos y manuales para ejecutar las pruebas.
[20]	Presenta una nueva manera de utilizar las ANN como *agentes* automatizados para probar un sistema.
[21]	Describe un modelo basado en ANN para identificar fallas cualitativas y cuantitativas en los módulos, como factor estadístico para predecir el nivel en que pueden aparecer en el producto completo.
[22]	Presentan *oráculos* de Redes Múltiples basados en ANN, con el objetivo de ejecutar automáticamente las pruebas.
[23]	Demuestra cómo incorporar las propiedades de ANN en las pruebas, dándoles significado mediante técnicas de aprendizaje a través de datos de entrada.
[24]	Propone aprovechar las ventajas de las ANN en el diseño de técnicas de prueba automatizadas.

Tabla 3. Integración de propiedades de la Inteligencia Artificial en las pruebas del software

Ref.	Descripción
[25]	Presenta una descripción de las características comunes entre AI e Ingeniería del Software y describe un acercamiento para su integración automática.
[26]	Analiza el papel de AI en la construcción de sistemas emuladoras del comportamiento humano, como base para la automatización de las pruebas.
[27]	Propone propiedades de AI como factor para maximizar la eficacia y minimizar los costos de la prueba automatizada.
[28]	Formula herramientas de AI como técnicas de prueba que pueden ayudar a mejorar la calidad del software.
[29]	Analiza el uso de AI en Ingeniería del Software y cubre las principales relaciones en las fases de desarrollo y sus métodos.
[30]	Explora las relaciones entre AI e Ingeniería del Software y establece desafíos para la automatización de las pruebas.
[31]	Encontraron que AI para las pruebas no solo reduce el costo, sino que garantiza una mejor calidad del producto y pruebas más exhaustivas.
[32]	Revisa las técnicas desarrolladas en AI desde el punto de vista de cómo utilizarlas en el diseño y automatización de los casos de prueba.
[33]	Propone un sistema de prueba usando técnicas de AI con el objetivo de reducir el conjunto de casos de prueba.
[34]	Se centra en las técnicas AI en la etapa conceptual, que se pueden asociar con la automatización de las pruebas.
[35]	Trabaja en cómo automatizar las pruebas aprovechando propiedades de AI.
[36]	Analiza las propiedades de AI y su relación para mejorar la calidad y confiabilidad del producto al automatizar la prueba.
[37]	Propone diversos enfoques de AI para optimizar los recursos de prueba en términos de su automatización.

2.2 Marco teórico

Dada la delimitación del alcance de esta investigación y a que el objetivo es encontrar interrelaciones entre las propiedades de VR, ANN y AI que permitan su utilización en la automatización de las pruebas del software, en las bases conceptuales no se tiene en cuenta cuestiones que se podrían considerar importantes, tales como Ingeniería del Software, Ingeniería

de Requisitos, Ciencias Computacionales, Métodos Formales y otros. A continuación, se describe los conceptos que, a juicio de los investigadores, era necesario consultar para alcanzar una comprensión del tema que se relaciona con la investigación y el contenido de este capítulo.

2.2.1 Pruebas del software

En un sentido amplio el software se prueba para determinar si cumple con los requisitos establecidos desde la primera fase del ciclo de vida, al tiempo que para establecer si el producto en desarrollo es el que solicita el cliente. Es decir, consiste en responder dos cuestiones: 1) estamos construyendo correctamente el sistema, y 2) estamos construyendo el sistema correcto; que se conocen ampliamente como actividades de V&V (Validación y Verificación) [38]. En el proceso se realiza una serie de actividades para descubrir y/o evaluar las propiedades de cada elemento del sistema, tales como planificación, preparación, ejecución, presentación de informes y gestión [39]. Por su parte, para Meyers [40] consiste en ejecutar un programa con la intención de encontrarle errores, mientras que Hass [41] afirma que es una actividad de soporte, porque no tiene sentido sin los procesos de desarrollo, además, porque no produce nada en sí misma: *si no hay nada desarrollado, no hay nada que probar*.

Es decir, la prueba no se trata de encontrar todos los errores que pueda tener el sistema, sino de descubrir situaciones que podrían afectar negativamente su funcionamiento [1, 2]. Sin embargo, hay que tener en cuenta que el costo de encontrar y corregir errores se puede elevar considerablemente durante el ciclo de vida. Por lo tanto, cuanto antes se descubran será mejor para controlar sus efectos, moderados o graves, en las etapas posteriores. La base de todo este proceso son los casos de prueba, descritos a través de atributos para determinar su eficacia y eficiencia para encontrar fallas; además, deben ser reutilizables, es decir, que se puedan modificar fácilmente para ejecutarlos en escenarios diferentes al original [42].

Al mismo tiempo, el proceso de detección de errores debe ser económico de realizar, analizar y depurar, ser evolutivo y requerir la mínima cantidad de esfuerzo. Hoy se acepta que la habilidad para realizar pruebas no consiste solamente en asegurar que los casos de prueba encuentren defectos, sino, además, que estén bien diseñados para evitar costos y tiempos excesivos [43].

2.2.2 Automatización de las pruebas

Aunque en la literatura se discute ampliamente acerca de la automatización de las pruebas, su definición todavía se concentra en temas como la creación de un mecanismo para automatizar casos de prueba, es decir, en la utilización de software especial para ejecutarlos a través de secuencias de comandos preparados, en lugar de un probador humano. En términos generales se asume que, con la automatización, las actividades manuales se sustituyen por sus equivalentes automatizadas [44], de tal manera que no se requiere la participación humana para ejecutar los casos de prueba. Por otro lado, la prueba automatizada consiste en utilizar un software especial para controlar la ejecución de los casos de prueba y, posteriormente, comparar los resultados reales con los resultados esperados [45].

Aunque algunas organizaciones han fracasado en su esfuerzo por implementarla y han tenido que recurrir nuevamente a los procesos manuales, a otras les ha permitido mejorar el rendimiento del equipo de trabajo e incrementar rápidamente la calidad del sistema. Por eso, para obtener los beneficios de la automatización, las pruebas deben ser cuidadosamente seleccionadas y aplicadas, porque la calidad del proceso es independiente de la calidad de la prueba y el hecho de ejecutarla manual o automáticamente no afecta ni su eficacia ni su evolución.

En tal sentido, no importa lo inteligente que se planee la automatización o lo bien que se ejecute, porque si la prueba en sí no logra nada, entonces, el resultado final será evidencia de que nada se logra al hacerlo de esa manera. Habitualmente, una vez implementada es más económica, porque el costo de funcionamiento es una fracción de los esfuerzos necesarios para hacerlo manualmente, sin embargo, en general cuesta más crearla y mantenerla. De ahí la importancia de seleccionar inteligentemente el momento para automatizar, porque será más barato ponerlo en práctica a largo plazo [46]; además, se debe tener en cuenta el mantenimiento, porque la sola actualización de un conjunto de casos de prueba automatizado puede tener un alto costo y no funcionar en otro escenario.

Con base en diferentes argumentos las empresas deben decidir entre automatizar o no automatizar, porque, aunque no se puede catalogar como la salvación para desarrollar sistemas de calidad, tampoco es una mala idea tenerla en cuenta. Para parte de la industria la automatización es una palabra que involucra mayor eficiencia, reducción de costos y entregas a tiempo, pero deben estar conscientes de que automatizar en sí no es el objetivo, sino lo que se puede obtener al aplicarla, es decir, el sistema es el objetivo final, por lo que la tarea es decidir cuál será el rol de la automatización para lograrlo. Además, debido a que la automatización genera y aplica casos de prueba a costos más bajos, con mejores resultados y con menos re-procesos, es lógico considerarla como beneficiosa. Lo contrario sería si generará re-procesos que exijan más probadores y un producto más costoso, porque entonces habría que considerarla inapropiada.

2.2.3 Realidad Virtual

En una amplia proporción los usuarios utilizan la tecnología de Realidad Virtual para interactuar en y con mundos y entornos simulados, en los que se genera una sensación de inmersión ilusoria para ellos. Pero existe otros usos, más orientados a la ingeniería, tales como en la mecánica, los materiales, la automatización industrial, la robótica, la arquitectura y la construcción [47]. En todo caso, el concepto de Realidad Virtual no es nuevo; el término ya era utilizado por Broderick [48] en su novela y, a pesar de haber permanecido por décadas como un pronóstico de la ciencia ficción, no fue sino hasta el siglo XXI que, tecnológicamente, se pudo materializar. Por eso, aunque desde un comienzo se observó su utilidad, hubo que esperar a que el desarrollo del hardware y el software permitieran su masificación para convertirla en una tecnología viable.

VR se ubica en la cima de las herramientas culturales que la humanidad ha empleado para plasmar, transmitir y experimentar sus ideas y, para muchos, es la propuesta más reciente en esta línea de tiempo. Algunas de esas herramientas se observan en los registros históricos y van desde pinturas, narraciones, experiencias, impresiones, ondas de radio y televisión, hasta la inmersión [4]. Debido a esto y aprovechando los avances en las Ciencias Computacionales, es una tecnología de simulación gráfica en tiempo real, que le permite al usuario experimentar la inmersión en una *realidad que no es su entorno natural*. En todo caso, el trabajo en VR ha generado principios útiles para la sociedad y la industria, tal como se evidencia en la investigación de Delaney [49].

Gracias a estos adelantos la sociedad comienza a migrar muchas de sus actividades a los mundos digitales, ocasionando al mismo tiempo que las nuevas generaciones abandonen cada vez más las interrelaciones físicas, para migrar a la inmersión en red. Parte de esas actividades están mediadas por tecnologías tales como teléfonos, video, mensajería, blogs, redes sociales, juegos, universos en línea, foros, canales de chat, … [50] que, en conjunto, conforman lo que algunos denominan como *ecología compleja* [51]. Una de las razones para esta rápida acogida de los universos virtuales es que en ellos desaparecen las normas, deberes y obligaciones del mundo real. Ya en el interior se debe respetar normas muy diferentes que, en la mayoría de casos, son

establecidas por los mismos usuarios, lo que genera fenómenos hasta ahora invisibles considerados como *discontinuos culturales*.

2.2.4 Redes Neuronales Artificiales

Las ANN se han convertido en una rama de estudio multidisciplinar en las ciencias en general, incluso desde antes de que se construyera el primer computador la ciencia estaba interesada en estudiarlas, y en la posibilidad de desarrollar modelos artificiales con diferentes aplicaciones. Para Kohonen [52] son redes de elementos simples interconectados masivamente en paralelo, usualmente adaptativos y con una organización jerárquica, que tratan de interactuar con los objetos del mundo real del mismo modo que lo hace el sistema nervioso biológico. Estas redes se comportan como un cerebro humano donde se procesa la información en paralelo, con la posibilidad de aprender y generalizar situaciones no incluidas en las vivencias diarias [53]. Este proceso se puede considerar como un método computacional orientado a resolver problemas complejos y con capacidad de realizar predicciones en sistemas relacionales no-lineales.

Los componentes arquitectónicos de las ANN son unidades computacionales cuyo funcionamiento se asemeja a las neuronas en el cerebro, y que se encargan de realizar cálculos individuales, pero que aportan al objetivo general de la red, ya sea en el aprendizaje o la capacitación [5]. Debido a su arquitectura algunos autores las consideran como un sistema masivo de procesamiento de información en paralelo, que utiliza control distribuido para aprender y almacenar conocimiento de su entorno [54]. Esta capacidad computacional la logran gracias a su diseño y a la capacidad de utilizar el conocimiento para encontrar resultados diferentes que, además, les permite funciones como resolver problemas complejos, extraer datos, reconocer patrones y aproximarse a funciones, en contextos diferentes al que les sirvió para aprender [55].

Ya que el funcionamiento de las Redes Neuronales Artificiales está determinado en gran medida por las conexiones entre sus elementos, es posible entrenarlas para ejecutar funciones particulares mediante ajustes a los valores de esas conexiones. De esta manera se pueden utilizar en técnicas de computación con el objetivo de derivar el significado de datos imprecisos, para encontrar patrones y para detectar tendencias complejas. Estas características han llevado a que autores como Kirkland y Wright [56] las denominen *sistemas expertos*, en el sentido de que se pueden utilizar para realizar proyecciones con base en situaciones complejas, además de responder preguntas con estructura: *qué pasaría si*.

Para aprovechar el máximo de su rendimiento las Redes Neuronales Artificiales primero se deben entrenar, es decir, modificar sus parámetros internos hasta llevar una función implementada lo más cerca posible de lo deseado. En otras palabras, se trata de optimizar el conjunto de parámetros de su estructura mediante ejemplos de capacitación, que les enseñan reglas subyacentes. Durante este proceso la red se adapta incrementalmente mediante conexiones que transportan información entre sus elementos y, al final, logra el aprendizaje con o sin supervisión. En el primer caso se necesita un entrenador que le indique la respuesta que debe entregar con base en la señal de entrada, para lo cual requiere información externa; mientras que en el segundo no requiere entrenador ni información externa, debido a que aprende con lo que encuentra localmente, lo que también se conoce como aprendizaje auto-organizado [57].

2.2.5 Inteligencia Artificial

Generalmente se acepta que como concepto y disciplina de investigación se originó en la conferencia de Dartmouth en 1956, y que desde entonces llamó la atención de científicos e

investigadores. Para Winston [58] la Inteligencia Artificial se refiere a los cálculos que hacen posible percibir, razonar y actuar; mientras que Wachsmuth [3] las identifica como un área de las Ciencias Computacionales que enfatiza en dichas acciones. Como campo de investigación hace énfasis en su poder de producir efectos o comportamientos inteligentes, por lo que busca progreso a través de sistemas que realizan síntesis antes que análisis. Wachsmuth sostiene que el objetivo de AI no es construir máquinas inteligentes con base en la inteligencia natural, sino comprender la inteligencia natural mediante la construcción de máquinas inteligentes, aunque para muchos, se trata es de buscar que las máquinas se comporten como en las películas. En todo caso, la línea principal la conforman dos capítulos: el científico y el ingenieril, con objetivos demarcados, aunque se confunden en cuanto a sus conceptos, métodos y herramientas [4].

Pero, a pesar del pesimismo inicial acerca de su utilidad y uso, la Inteligencia Artificial mantiene hoy la promesa de potencializar los desarrollos de las Ciencias Computacionales para abrir nuevos caminos a la ciencia, resolver problemas complejos y mejorar la calidad de los sistemas y, para cumplir esa promesa, debe interactuar con disciplinas tales como la Ingeniería del Software, en entornos de soporte, herramientas, técnicas y uso de hardware y software [59]. Aunque algunos son escépticos acerca de este tipo de interacción [60], la realidad es que la AI ha abierto un nuevo campo de investigación en las Ciencias Computacionales, y en los últimos años ha logrado alto impacto con sus desarrollos.

Por otro lado, la imagen general de AI ha hecho que, por décadas y con matices de imaginación, esperanza y temor, se hable de un futuro dominado por máquinas inteligentes. Pero dejando de lado estas creencias hay que aceptar que las técnicas de AI predominarán en los desarrollos IT de las próximas décadas. Algo que se empieza a materializar con iniciativas como Partnership on AI, en la que participan Amazon, Google, Facebook, IBM y Microsoft bajo el lema *AI en beneficio de las personas y la sociedad*. Los resultados venideros se observarán desde dos perspectivas: 1) la que imita competencias humanas únicas, tales como el reconocimiento de letras o la generación de voz, con la que se cubre la mayoría de las aplicaciones actuales; y 2) la que se refiere a sistemas que realizan actividades creativas, con espontaneidad, sin supervisión ni solicitud, y que se podrían caracterizar como *conscientes*. Esta perspectiva todavía no se desarrolla completamente, pero es probable que se logre en poco tiempo [61].

Esta perspectiva ha hecho que, desde ya, se empiece a anhelar sistemas inteligentes que realicen análisis y tomen decisiones con base en datos que cambian rápidamente, que busquen patrones y que se encarguen de tareas sensibles que exigen tiempo y cuidado. La esperanza es que le den sentido al alto volumen de información que circula hoy, que generen conocimiento y que se conviertan en una especie de *oráculo*, al que se pueda consultar en busca de recomendaciones acerca de *qué hacer* [62]. Esta esperanza es la que hace tan popular a la Inteligencia Artificial y se ha convertido en el motor que presiona a la ciencia y la ingeniería por resultados a corto plazo. Porque en el fondo se espera que, como ciencia, logre que los computadores ejecuten tareas, procesen información inteligentemente y que aprendan y se adapten a los cambios de cada contexto, tal como lo haría un humano [63].

2.3 Integración de VR, ANN y AI en la automatización de las pruebas

En varios trabajos de los investigadores, correspondientes a la primera fase de la investigación de la que hace parte este capítulo, se identificaron las propiedades más significativas de la Realidad Virtual [4], de las Redes Neuronales Artificiales [5] y de la Inteligencia Artificial [6]. Posteriormente, y con base en la escala de valoración: *Muy débil*, *Débil*, *Medio*, *Fuerte* y *Muy fuerte*, se estimó la probabilidad de integrarlas en la automatización de las pruebas del software. En las Tablas 4, 5 y

6 se presenta la valoración promedio obtenida a partir de los resultados de la revisión, las apreciaciones de los especialistas y la experiencia de los investigadores.

Tabla 4. Valoración a la probabilidad de integrar propiedades de VR en la automatización delas pruebas

Fuente	Propiedad [4]	Probabilidad de uso en la automatización				
		Muy débil	Débil	Medio	Fuerte	Muy fuerte
[10, 63]	Virtuality					■
[64-67]	Modeling and Simulation					■
[13, 68]	Interactivity					■
[9, 11]	Immersion				■	
[69]	Sensoriality		■			
Experiencia de los autores	Multidimensionality				■	
[14]	Dynamism				■	
[70, 71]	Multimediality		■			
[8]	Multiplicity					■
[72]	Flexibility				■	
[73]	Immateriality			■		

Tabla 5. Valoración a la probabilidad de integrar propiedades de ANN en la automatización de las pruebas

Fuente	Propiedad [5]	Probabilidad de uso en la automatización				
		Muy débil	Débil	Medio	Fuerte	Muy fuerte
[23]	Adaptive learning					■
[74, 75]	Self-organization					■
[21, 76]	Fault tolerance				■	
[74]	Real-time operation					■
[17, 75]	Expansion			■		
[74]	Convergence			■		
[17, 20]	Validation		■			
Experiencia de los autores	Complexity			■		
[20, 21]	Non-linearity		■			
[23, 24]	Training		■			
[18, 77, 78]	Memorization					■
[79, 80]	Agility				■	

Tabla 6. Valoración a la probabilidad de integrar propiedades de AI en la automatización de las pruebas

Fuente	Propiedad [6]	Probabilidad de uso en la automatización				
		Muy débil	Débil	Medio	Fuerte	Muy fuerte
[31]	Interactivity					■
[37]	Real-time response					■
[32, 81]	Cognitive autonomy					■
[82, 83]	Decision making					■
[25, 31]	Multitask				■	
[37, 84]	Multiple systems				■	
[32, 81]	Autonomous Learning					■
[36, 82]	Memorization					■
[85]	Decision making				■	
Experiencia de los autores	Rational thinking				■	
[84]	Creation and execution			■		
[35, 82]	Heuristic search		■			

3. ANÁLISIS Y DISCUSIÓN

Los sistemas software se estructuran mediante componentes interrelacionados que, utilizando diversos medios de comunicación, intercambian información desde y hacia diferentes dimensiones y plataformas. Esta lógica distribuida en capas ha incrementado el nivel de complejidad de las pruebas antes de ponerlos en funcionamiento, haciendo que los equipos de

desarrollo tengan que asegurarse de incluir las pruebas como un proceso paralelo e integrarlas al ciclo de vida desde el inicio. Como se ha tratado previamente, para el software ya no es suficiente con aplicar pruebas manuales, y la realidad es que se necesita automatizar el proceso en un alto porcentaje, de tal manera que se pueda identificar los problemas funcionales y estructurales del sistema, antes de que entre en funcionamiento. Por lo que el objetivo para el equipo de desarrollo debe ser el de alcanzar una visión integral de calidad, fiabilidad y seguridad del producto antes de entregarlo al cliente, y corregir las fallas antes que las descubra el usuario.

De acuerdo con lo encontrado en esta investigación las herramientas para automatizar las pruebas les ayudan en gran medida a los desarrolladores y a los arquitectos de software para superar estos desafíos. Pero, aunque muchas abordan las complejas necesidades de la prueba, todavía no se puede afirmar que encuentran y resuelven la mayoría de los problemas, o que estructuran e implementan un adecuado conjunto de casos de prueba. Las razones para esto son muchas, pero en los resultados de este trabajo se ha determinado que una de las más influyentes, dado el tipo de sistemas y los problemas que solucionan, es que han sido pensadas y construidas con un enfoque disciplinar. De esta manera puede que respondan adecuadamente en un determinado tipo de pruebas, para un sistema específico, pero no para otro tipo que tiene mayor importancia en otros sistemas.

La propuesta es integrar propiedades desde disciplinas que han demostrado utilidad y aplicabilidad en las pruebas, y construir una herramienta automatizada para diseñar, implementar, analizar y actualizar los casos de prueba, de acuerdo con el tipo de prueba (funcional o estructural) que se desea aplicar, y con el tipo de sistema que se desea probar. Es decir, que se auto-delimite, auto-dimensione y auto-programe para atender las solicitudes de prueba, sin importar el sistema sobre el que se aplique. En lo descrito en este capítulo, y de acuerdo con los resultados obtenidos en la investigación, la propuesta se estructura de la siguiente manera:

1. Tomar las propiedades de la Realidad Virtual, las Redes Neuronales Artificiales y la Inteligencia Artificial, valoradas como Fuerte o Muy fuerte, con mayores probabilidades de integrarse en una herramienta de este tipo, y definirlas con criterios que permitan interrelacionarlas en una solución automatizada. Si las propiedades se repiten en varias de estas disciplinas, solamente se toma una como referencia. En el caso de VR las seleccionadas son [4]:

   - *Inmersión*. Se refiere a un estado del yo en el que la conciencia se compenetra en un mundo virtual absorbente, que representa la simulación de uno real.
   - *Multidimensionalidad*. Los mundos virtuales deben reflejar con exactitud las dimensiones reales, para que el cerebro asimile adecuadamente su estadía física dentro de algo no-físico.
   - *Dinamismo*. Es la capacidad de transformación y adaptación, que tienen los elementos de un mundo virtual, para que el usuario experimente mayor nivel de realidad en relación con el que obtendría en un mundo estático.
   - *Flexibilidad*. Es la facultad de los elementos de un entorno para adaptarse fácilmente a las necesidades del diseño, objetivo y nivel de inmersión que se desea recrear en el mundo virtual.
   - *Virtualidad*. Es el contenido de un mundo dado que puede existir únicamente en la mente del autor, o que puede compartir con otros.
   - *Modelado y Simulación*. El modelo representa al sistema en sí y la simulación su funcionamiento en el tiempo, y el objetivo es visualizar los efectos reales eventuales de las condiciones alternativas en el funcionamiento de éste, para seleccionar cursos de acción.

- *Interactividad*. Es el grado en que la tecnología crea entornos en los que los usuarios se comunican, síncrona o asincrónicamente, e interactúan a través de un medio.
- *Multiplicidad*. Se relaciona con la presencia y el espacio en el sistema virtual, donde la duración de las progresiones y las interrelaciones provienen desde múltiples puntos y a través de múltiples medios, mediados por una idea geométrica.

Las propiedades de las Redes Neuronales Artificiales seleccionadas son las siguientes [5]:

- *Tolerancia a fallos*. Capacidad de un sistema para continuar en funcionamiento, aunque falle alguno o algunos de sus componentes.
- *Agilidad*. Habilidad de un sistema para cambiar eficazmente su ubicación y posición, manteniendo control total de sus componentes, para adaptarse a un cambio en el contexto.
- *Aprendizaje adaptativo*. Capacidad para aprender de los contextos desarrollando agilidad para adaptarse a cada uno.
- *Auto organización*. Capacidad que le permite al sistema, después de haber aprendido los patrones de un dominio, reconocer otros similares, aunque no se haya entrenado para ellos.
- *Operación en tiempo real*. Habilidad de un sistema para reconocer patrones en tiempo real y de actualizar simultáneamente todos sus componentes, debido a que trabajan en paralelo.
- *Memorización*. Aunque es una actividad intelectual humana, en los sistemas consiste en ubicar y guardar patrones, datos y secuencias que consulta cuando sea necesario.

Por último, las propiedades de la Inteligencia Artificial seleccionadas son [6]:

- *Multitarea*. Es la propiedad de los sistemas de permitir o ejecutar varios procesos simultáneamente, al a vez que comparten recursos, datos, patrones y secuencias.
- *Sistemas múltiples*. Es la habilidad de un sistema para interrelacionar las funciones de diversos sistemas, semejantes o no, para lograr un objetivo determinado.
- *Autonomía cognitiva*. Se refiere a las funciones, procesos y estados que les permite a los sistemas comprender, inferir, tomar decisiones, planificar y aprender con base en la información que procesan.
- *Toma de decisiones*. Es un proceso que ejecutan los sistemas para definir, recopilar y procesar datos, generar información, inferir alternativas y seleccionar una con base en la identificación más apropiada para resolver un problema específico.
- *Aprendizaje autónomo*. Es la capacidad de un sistema para auto-dirigirse, regularse, actualizarse y definirse, para estructurar e instalar una solución basada en su aprendizaje.

2. Integrar las propiedades en la automatización de alguno de los tipos de pruebas (funcionales o estructurales) o en ambas. Esta integración se decide de acuerdo con las definiciones y recomendaciones de los autores, los especialistas consultados y la experiencia de los investigadores y, para esta investigación el resultado se observa en la Figura 1. Los criterios para realizar esta integración se desprenden de: 1) el análisis a los resultados de los trabajos seleccionados en la búsqueda, teniendo en cuenta que se obtuvieron utilizando solamente propiedades de cada área por separado, pero que el contexto en el que se experimenta es la automatización de las pruebas; 2) las recomendaciones de los especialistas, definidas en las entrevistas y diálogos, sobre las propiedades que aceptan como útiles en un proceso de integración para la automatización; y 3) la experiencia de los investigadores, acumulada en sus investigaciones sobre cómo solucionar el problema de la automatización de las pruebas.

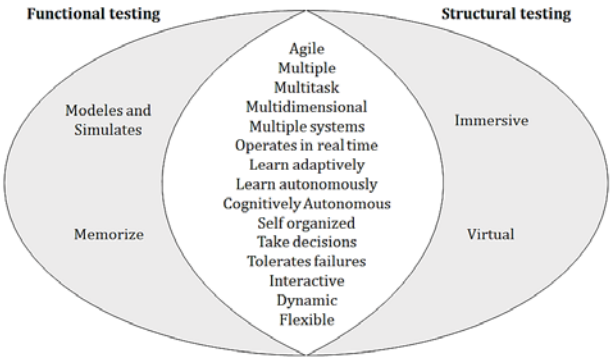

Figura 1. Propiedades de VR, ANN y AI que se podrían integrar en la automatización de las pruebas

Se observa que la mayoría de las propiedades evaluadas se ubican en la *zona común* para las pruebas funcionales y las estructurales, lo que indica que existe una amplia posibilidad de integrarlas para el logro del objetivo de la investigación. Esta característica les permite a los desarrolladores adquirir el conocimiento necesario para optimizar el código y, por lo tanto, el tiempo y los costos necesarios para la estructuración, el diseño y el desarrollo de las herramientas. Además, para proyectar las actualizaciones y las innovaciones que el producto final necesitará en el transcurso de su vida útil, porque pueden asimilar cada propiedad como un requisito en el ciclo de vida y, cuando se modifican las propiedades, también lo harán los requisitos.

3. Estructurar la herramienta que se puede desarrollar para atender estos *requisitos*, teniendo en cuenta los objetivos y características de las pruebas funcionales y las estructurales. Aquí se recomienda las siguientes alternativas:

   - Para las pruebas estructurales, cuyo objetivo es correr el sistema para evaluar su funcionamiento: agilidad, seguridad y fiabilidad, además de capacidades tales como velocidad de respuesta, robustez, integralidad y manejo de los datos, y que, por definición, las ejecuta una persona a partir de un conjunto de casos de prueba. La propuesta es construir un *robot* que la reemplace. Esta herramienta se diseña a partir de los requisitos establecidos previamente, teniendo en cuenta la estructura, la semántica y la gramática del lenguaje de programación con el que se desarrolla el sistema. El robot analiza la información que recolecta de cada contexto de programación y de cada dominio del sistema, para conformar un conjunto de casos de prueba que aplica bajo tales circunstancias. Luego compara los resultados obtenidos con los esperados y presenta un reporte de la prueba, indicando las fallas encontradas.

   - De acuerdo con la estructura interna definida por la gramática y la semántica del lenguaje utilizado, el objetivo de las pruebas funcionales es revisar el código para determinar errores de procedimiento, además de la lógica matemática de las ecuaciones, funciones o fórmulas incluidas. Se diferencian de las funcionales en que aquí no importa el funcionamiento del sistema, sino que la lógica del código permita el logro de los objetivos del mismo. Entonces, la herramienta recomendad debe ser una *máquina virtual* que reemplace a la persona encargada de leer el código y que, normalmente, es el mismo desarrollador. Aprovechando las propiedades ubicadas en la Figura 1 para las pruebas estructurales y comunes, esta máquina recorre el código analizando sus procedimientos, a la vez que interactúa con el robot de las pruebas funcionales para determinar si las fallas, que éste encuentra en el funcionamiento del sistema, se deben a errores de código. En este caso muestra virtualmente la línea del error y continúa interactuando con él mientras duren las pruebas, y hará lo mismo cuando el error se genera por procedimientos en el código.

Este trabajo multitarea e interrelacional de las herramientas les permite a ambos componentes adquirir autonomía cognitiva, a la vez que aprendizaje autónomo y adaptativo, con lo que pueden tomar decisiones dinámicas, flexibles e interactivas para solucionar las fallas, a la vez que memorizar los procedimientos de respuesta ante cada contexto. Por un lado, el robot opera en tiempo real y adecua el conjunto de casos de prueba a cada sistema que debe probar y, por otro lado, la máquina virtual auto-organiza los errores más comunes en el código con base en el lenguaje utilizado. Estos multiprocesos enriquecen el acervo cognitivo de las herramientas mediante intercambio de información, que cada una convierte en conocimiento para ejecutar sus funciones de la manera más adecuada en cada situación de prueba.

4. CONCLUSIONES

La industria del software, la sociedad y los clientes se hallan en una encrucijada: por un lado, para garantizar la calidad, seguridad y fiabilidad de este desarrollo tecnológico, y antes de implementarlo y ponerlo en funcionamiento, es necesario aplicar un adecuado plan de pruebas; por otro lado, la forma como se estructuran, diseñan y ejecutan las pruebas consume tiempo, recursos e incrementa el costo final del desarrollo. Una alternativa consiste en automatizar las pruebas, lo más que se pueda, e involucrarlas como una actividad paralela, para que no se reduzca a un proceso al final del ciclo de vida. Algunos autores han experimentado diversas maneras de alcanzar este objetivo, pero, como muchos de ellos lo aceptan, su trabajo se queda corto, porque no lo hacen de forma transdisciplinar. Es decir, enfocan la solución desde una mirada disciplinar y utilizando las propiedades que mejor se ajusten al objetivo del trabajo.

En esta investigación se asume que la automatización de las pruebas del software se podría lograr a través de un análisis e integración transdisciplinar de propiedades desde diversas áreas, cuya funcionalidad ya ha sido experimentada y validada por separado. La meta trabajo es proponer, desde una concepción teórica, que existe propiedades de la Realidad Virtual, las Redes Neuronales Artificiales y la Inteligencia Artificial que se pueden interrelacionar e integrar en la automatización de las pruebas, a través del diseño de herramientas orientadas a las pruebas funcionales y estructurales, pero con visión transdisciplinar.

Los desafíos que este acercamiento teórico genera van desde la posibilidad de materializarlo en la práctica, con el desarrollo de las herramientas, hasta validar y verificar que realmente funcionan, pero los investigadores están convencidos de que con los descubrimientos e innovaciones [86] que se han alcanzado, especialmente en VR, ANN y AI, ya es momento de iniciar procesos conducentes al logro. Las descripciones que se presentan en los resultados de este trabajo son la base ideológica para otras iniciativas, que le den continuidad a través de una labor de comunidad y con el objetivo de materializarlos.

El trabajo futuro más importante que se proyecta a partir de los resultados presentados es la conceptualización, el diseño y el desarrollo del robot y de la máquina virtual. En este sentido, se necesita conformar un equipo amplio, transdisciplinar y sin estrechez mental, que asuma el desafío como un producto alcanzable y factible de construir con lo que se conoce y se espera. Además, los autores esperan que este trabajo llame la atención de todos los interesados en este tipo de retos y que se animen a conformar el quipo mencionado.

REFERENCIAS

[1] Serna E. (2013). Functional Test of Software - A Constant Verification Process. Fondo Editorial ITM.

[2] Serna E. y Arango F. (2010). Effectiveness analysis of the set of test cases generated with the requirements by contracts technique. En V Cong. Colombiano de Computación. Cartagena, Colombia.

[3] Wachsmuth I. (2000). The concept of intelligence in AI. En H. Cruse et al. (Eds.), Prerational Intelligence - Adaptive Behavior and Intelligent Systems without Symbols and Logic. Kluwer Academic Publishers.
[4] Serna E. et al. (2020). Principles of Virtual Reality as cultural aspects seen from Social Relations. Revista de Ciencias Sociales. In press.
[5] Acevedo E. et al. (2017). Principles and characteristics of artificial neural networks. En E. Serna (Ed.), Desarrollo e Innovación en Ingeniería (pp. 173-182). Editorial Instituto Antioqueño de Investigación.
[6] Serna A. et al. (2017). Principles of artificial intelligence in computer science. En E. Serna (Ed.), Desarrollo e Innovación en Ingeniería (pp. 161-172). Editorial Instituto Antioqueño de Investigación.
[7] Bierbaum A. y Just C. (1998). Software tools for virtual reality application development. En Course Notes for SIGGRAPH 98 Course 14. Applied Virtual Reality.
[8] Bierbaum A et al. (2003). Automated testing of virtual reality application interfaces. En 9th workshop on virtual environments. Zurich, Sswitzerland
[9] Dwivedi S. et al. (2003). Beta testing of design product using virtual reality simulation. Intl. Journal of Agile Manufacturing 34, 45-60.
[10] Foit K. (2007). Introduction to use virtual reality visualisations in the exploitation and virtual testing of machines. J. Achieve Mater Manufact Eng. 25(2), 57-60.
[11] Zhao X. et al. (2007). Software testing applications based on a virtual reality system. J. Electron Sci Technol China 5(2), 120-124.
[12] Steed A. (2008). Proposals for future virtual environment software platforms. En S. et al. (Eds.), Virtual Realities (pp. 56-76). Springer.
[13] Avgoustinov N. et al. (2011). Virtual reality in planning of non-destructive testing solutions. En A. Bernard (Ed.), Global Product Development (pp. 705-710). Springer.
[14] Falcão C. y Soares M. (2013). Application of virtual reality technologies in consumer product usability. Lecture Notes on Computer Science 2013, 342-351.
[15] Khoshgoftaar T. et al. (1997). Application of neural networks to software quality modeling of a very large telecommunications system. IEEE Trans Neural Netw 8(4), 902-909.
[16] Dawson C. (1998). An artificial neural network approach to software testing effort estimation. Trans Info Comm Technol. 20, 1-11.
[17] Vanmali M. et al. (2002). Using a neural network in the software testing process. Int. Sys. 17(1), 45-62.
[18] Mao Y. et al. (2006). Neural networks based automated test oracle for software testing. Lect Notes Comp Sci. 4234, 498-507.
[19] Smilgyte K. y Nenortaite J. (2011). Artificial neural networks application in software testing selection method. Lect Notes Comp Sci. 6678, 247-254.
[20] John J. (2011). A performance based study of software testing using artificial neural network. Int J Logic Based Intel Syst. 1(1), 45-60.
[21] Sandhu P. et al. (2012). Neural network approach for software defect prediction based on quantitative and qualitative factors. Int J Comp Theory Eng. 4(2), 298-303.
[22] Shahamiri S. et al. (2012). Artificial neural networks as multi-networks automated test oracle. Auto Softw Eng. 19(3), 303-334.
[23] Yogi A. (2013). Reformation with neural network in automated software testing. Int J Comp Trends Technol. 4(6), 1816-1819.
[24] Sathyavathy V. (2017). Evaluation of software testing techniques using artificial neural network. Int J Eng Comp Sci. 6(3), 20617-20620.
[25] Rech J. y Althoff K. (2004). Artificial intelligence and software engineering: Status and future trends. Künstliche Intelligenz 18(3), 5-11.
[26] Zeigler B. et al. (2009). Artificial intelligence in modeling and simulation. En R. Meyers (Ed.), Encyclopedia of Complexity and System Science (pp. 344-368). Springer.
[27] Staats M. (2010). The influence of multiple artifacts on the effectiveness of software testing. En IEEE/ACM international conference on automated software engineering. Antwerp, Belgium.
[28] Larkman D. et al. (2010). General application of a decision support framework for software testing using artificial intelligence techniques. En Second KES International Symposium. Baltimore, USA
[29] Meziane F. y Vadera S. (2010). Artificial intelligence in software engineering - Current developments and future prospects. En F. Meziane y S. Vadera (Eds.), Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects (pp. 278-299). Information science reference.

[30] Harman M. (2012). The role of artificial intelligence in software engineering. En First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. Zurich, Switzerland.
[31] Rauf A. y Alanazi M. (2014). Using artificial intelligence to automatically test GUI. En 9th International Conference on Computer Science & Education. Vancouver, Canada.
[32] Shankari K. y Thirumalaiselvi R. (2014). A survey on using artificial intelligence techniques in the software development process. Int J Eng Res Appl. 4(12), 24-33.
[33] Kire K. y Malhotra N. (2014). Software testing using intelligent technique. Int J Comp Appl. 90(19), 22-25.
[34] Sharma S. y Pandey K. (2015). Integrating AI techniques in SDLC. En Third International Symposium on Women in Computing and Informatics. Kochi, India.
[35] Pawar P. (2016). Application of artificial intelligence in software engineering. J Comp Eng. 18(3), 46-51.
[36] Bhateja N. (2016). Various artificial intelligence approaches in field of software testing. Int J Comp Sci Mobile Comput. 5(5), 278-280.
[37] Bhateja N. y Sikka S. (2017). Achieving quality in automation of software testing using AI based techniques. Int J Comp Sci Mobile Comput. 6(5), 50-54.
[38] Boehm B. (1989). Software risk management. Lect Notes Comp Sci. 387, 1-19.
[39] ISO/IEEE. (2013). ISO/IEEE 29119 - Part I International Standard. Software and systems engineering/ software testing, concepts and definitions. IEEE.
[40] Myers J. (1979). The Art of Software Testing. Wiley.
[41] Hass A. (2008). A Guide to Advanced Software Testing. Artech House.
[42] Karthikeya S. y Rao S. (2014). Adopting the right software test maturity assessment model. En Cognizant 20-20 Insights. New Jersey, USA.
[43] Anya P. y Smith G. (2014). Qualitative research methods in software engineering. Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 4(2), 14-18.
[44] Haugset B. y Hanssen G. (2008). Automated acceptance testing: A literature review and an industrial case study. En Agile Development Conference. Toronto, Canada.
[45] Safronau V. y Turlo V. (2011). Dealing with challenges of automating test execution. En Third International Conference on Advances in System Testing and Validation Lifecycle. Barcelona, Spain.
[46] Furtado A. et al. (2014). Towards a maturity model in software testing automation. En The Ninth International Conference on Software Engineering Advances. New York, USA.
[47] Mujber T. et al. (2004). Virtual reality applications in manufacturing process simulation. J. Mater Process Technol. 1834, 155-156.
[48] Broderick D. (1982). The Judas Mandala. Pocket Books.
[49] Delaney N. (2004). The market for visual simulation/virtual reality systems. Cyberedge.
[50] Hernández A. y Pérez K. (2017). Criteria for verifying and validating mechanisms in the development of videogames. Revista Antioqueña de las Ciencias Comput. y la Ingeniería de Software 7(1), 7-12.
[51] Kow Y. y Young T. (2013). Media technologies and learning in the StarCraft eSport community. En Conference on Computer Supported Cooperative Work. San Antonio, USA.
[52] Kohonen T. (1988). An introduction to neural computing. Neural Netw.1(1), 3-16.
[53] Singhal D. y Swarup K. (2011). Electricity price forecasting using artificial neural networks. Int J Electric Power Energy Syst. Elsevier 33(3), 550-555.
[54] Haykin S. (1999). Neural Networks. A Comprehensive Foundation. Prentice Hall.
[55] Kulkarni A. (1993). Artificial Neural Networks for Image Understanding. John Wiley.
[56] Kirkland L. y Wright R. (1997). Using a neural network to solve testing problems. IEEE Aerospace Electron Syst Magazine 12(8), 36-40.
[57] Maimon O. y Last M. (2000). Knowledge Discovery and Data Mining - The Info Fuzzy Network (IFN) Methodology. Kluwer Academic Publishers.
[58] Winston P. (1993). Artificial Intelligence. Addison-Wesley.
[59] Partridge D. (1992). The relationships of AI to software engineering. En IEEE Colloquium on Software Engineering and AI. London, UK.
[60] Last M. et al. (2004). Artificial Intelligence Methods in Software Testing. World Scientific Publishing.
[61] Mair C. et al. (2000). An investigation of machine learning based prediction systems. J Syst Softw. 53(1), 23-29.
[62] Aggarwal K. et al. (2004). A neural net based approach to test oracle. ACM Softw Eng Notes 29(3), 1-6.

[63] Gelly S. et al. (2012). The grand challenge of computer Go: Monte Carlo tree search and extensions. Comm ACM. 55(3), 106-113.
[64] Alsmadi I. (2012). How much automation can be done in testing? En I Alsmadi (Ed.), Advanced Automated Software Testing: Frameworks for Refined Practice (pp. 1-29). Information Science.
[65] Stone B. y Pegman G. (1995). Robots and virtual reality in the nuclear industry. Serv Robot 1(2),24-27.
[66] Stuart R. (1996). The Design of Virtual Environments. McGraw-Hill.
[67] Gong Y. et al. (2002). The simulation of grinding wheels and ground surface roughness based on virtual reality technology. J. Mater Process Technol. 129(1-3), 123-126.
[68] Foit K. et al. (2006). The project of an off-line, remote programming system for Mitsubishi Movemaster industrial robot. En XIII international scientific and technical conference. Sevastopol, Ukraine.
[69] Kuliga S. et al. (2015). Virtual reality as an empirical research tool - Exploring user experience in a real building and a corresponding virtual model. Comp. Environm Urban Syst. 54, 363-375.
[70] Gausemeier J. et al. (2010). Virtual and augmented reality for systematic testing of self-optimizing systems. En International Design Conference. Dubrovnik, Croatia.
[71] Godoy J. (2012). Have the software testing a future? Revista Antioqueña de las Ciencias Computacionales y la Ingeniería de Software 2(1), 18-25.
[72] Hackett M. (203). The changing landscape of software testing. LogiGear Magazine.
[73] Cigniti. (2002). How to amplify the impact of virtual reality with a robust test strategy? Cigniti Technologies.
[74] Banerjee P. (2009). Virtual reality and automation. En S. Nof (Ed.), Springer Handbook of Automation (pp. 269-278). Springer.
[75] John J. y Wanjari M. (2014). Performance based evaluation of new software testing using artificial neural network. Int J Sci Res. 3(5), 1519-1534.
[76] Shahamiri S. y Nasir W. (2008). Intelligent and automated software testing methods classification. En 4th postgraduate annual research seminar. Skudai, Malaysia.
[77] Saraph P. et al. (2004). Test set generation and reduction with artificial neural networks. En M. Last et al. (Eds.), Artificial Intelligence Methods in Software Testing (pp. 101-132). World Scientific.
[78] Ye M. et al. (2006). Neural networks based automated test oracle for software testing. En 13th international conference on Neural information processing. Hong Kong, China
[79] Su Y, Huang C. Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models. J Syst Softw. 2007;80(4):606-615.
[80] Kumar A. (2005). Dynamic test case generation using neural networks. Report for CS497. Indian Institute of Technology.
[81] Kire K. (2014). Software testing using intelligent technique. Int J Comp Appl. 90(19), 22-25.
[82] Moore A. et al. (2017). Automation and AI - From man to machine. Computer Science.
[83] Platz W. (2017). Beyond continuous testing with artificial intelligence. Tricentis.
[84] Nordström J. (2016). The Future Will Be Automated. 3GAMMA.
[85] Matthews K. (2017). How machine learning is impacting the way we test software. TechZone 360°.
[86] Roldán M. et al. (2016). State of the art and methodological approach to product innovation evaluation in organizations of the telecommunications industry. Revista Actas de Ingeniería 2, 210-218.

Métodos Formales, Ingeniería de Requisitos y Pruebas del Software

El libro es sí es una recopilación de los trabajos del profesor Edgar Serna y su equipo, quienes han dedicado gran parte de su investigación a aportar al logro de la meta de mejorar los productos software, y que, si bien han sido publicados en revistas o presentados en Congresos, reunirlos en un texto completo es una apuesta para que los ingenieros de software tengan herramientas más cercanas. Se trata de una iniciativa en la que se reúne la producción de estos investigadores a partir de un programa de investigación estructurado y ejecutado por más de dos décadas.

Editorial Instituto Antioqueño de Investigación