Model-based testing using UML activity diagrams: A systematic mapping study

Tanwir Ahmad*, Junaid Iqbal*, Adnan Ashraf, Dragos Truscan, Ivan Porres

Faculty of Science and Engineering, Åbo Akademi University, Vesilinnantie 3, 20500 Turku, Finland

Abstract

Context: The Unified Modeling Language (UML) has become the de facto standard for software modeling. UML models are often used to visualize, understand, and communicate the structure and behavior of a system. UML activity diagrams (ADs) are often used to elaborate and visualize individual use cases. Due to their higher level of abstraction and process-oriented perspective, UML ADs are also highly suitable for model-based test generation. In the last two decades, different researchers have used UML ADs for test generation. Despite the growing use of UML ADs for model-based testing, there are currently no comprehensive and unbiased studies on the topic.
Objective: To present a comprehensive and unbiased overview of the state-of-the-art on model-based testing using UML ADs.
Method: We review and structure the current body of knowledge on model-based testing using UML ADs by performing a systematic mapping study using well-known guidelines. We pose nine research questions, outline our selection criteria, and develop a classification scheme.
Results: The results comprise 41 primary studies analyzed against nine research questions. We also highlight the current trends and research gaps in model-based testing using UML ADs and discuss some shortcomings for researchers and practitioners working in this area. The results show that the existing approaches on model-based testing using UML ADs tend to rely on intermediate formats and formalisms for model verification and test generation, employ a multitude of graph-based coverage criteria, and use graph search algorithms.
Conclusion: We present a comprehensive overview of the existing approaches on model-based testing using UML ADs. We conclude that (1) UML ADs are not being used for non-functional testing, (2) only a few approaches have been validated against realistic, industrial case studies, (3) most approaches target very restricted application domains, and (4) there is currently a clear lack of holistic approaches for model-based testing using UML ADs.

Keywords: Systematic mapping study, UML activity diagram, Software testing, Test generation, Model-based testing

1. Introduction

Software-based systems are permeating the ever-increasing number of business areas starting from web applications to systems used to manage critical functions of different devices and safety-critical systems, for instance, pacemakers, aircraft, and nuclear power plants. Therefore, such systems must meet the quality criteria required by the application domain and expected by their customers. Software testing is the process of observing and demonstrating that the behavior of a software system conforms with its specifications [1].

*Corresponding authors.
Email addresses: `tanwir.ahmad@abo.fi` (Tanwir Ahmad), `junaid.iqbal@abo.fi` (Junaid Iqbal), `adnan.ashraf@abo.fi` (Adnan Ashraf), `dragos.truscan@abo.fi` (Dragos Truscan), `ivan.porres@abo.fi` (Ivan Porres)

This process evaluates the quality of the product and finds potential failures in the System Under Test (SUT). The software testing process can be segmented broadly into three stages: *creating test cases, executing test cases, and evaluating test cases* [2]. A *test case* can be defined as a sequence of input and expected output values [1]. There are two different software testing techniques that are used to create test cases: *white-box or structural testing*, where test cases are based on the source code of the SUT [3], and, *black-box or specification based testing*, where test cases are constructed from the specification of the SUT.

One of the black-box software testing techniques that improve the traditional software testing process with automatic test generation is Model-Based Testing (MBT) [4]. During the last two decades, practitioners have used MBT in various software domains because it allows for early testing of the SUT and for automating or semi-automating the entire testing process. In MBT, abstract models are built to capture the behavior of the SUT and, later on, to generate test cases [5]. These models can be created from requirements expressed in natural language or by considering other software artifacts such as design models.

Since the main intent of most of the software testing activities is to find faults which occur during the execution of the system, behavioral models which represent system behavior are highly suitable for test case generation [6]. There are several types of behavioral models that are typically used for test case generation, for example, Finite State Machines (FSM) [7], StateCharts [8], Petri nets [7], and Markov Chain Models [9]. Owing to the increasing popularity of the Unified Modeling Language (UML) [10], UML state machine diagrams have been used extensively for MBT because most of the software solutions are state-based. However, in recent years, the UML Activity Diagrams (ADs), which traditionally have been used for documenting business use cases, have started to become popular for test case generation, especially for acceptance testing because ADs can be used to effectively model the complex work flow of business systems [11]. However, despite the growing use of UML ADs for MBT, there are currently no comprehensive and unbiased studies on the topic. Therefore, there is a need to summarize the existing approaches and identify the research gaps and requirements for future studies.

To identify and characterize the existing MBT initiatives that use UML ADs, we have performed a Systematic Mapping Study (SMS) [12, 13]. An SMS is a methodology to objectively collect and classify the primary research papers in a specific research area. From the existing literature on UML based test case generation, one could find three literatures reviews [14, 15, 16]. The objective of these reviews partially overlap with our study. However, none of these studies is systematic or comprehensive. Kaur and Vig [17] presented a Systematic Literature Review (SLR) on test case generation from UML models, but they did not follow the SLR guidelines outlined by Kitchenham and Charters [18]. Moreover, their results include only nine papers on UML ADs.

To the best of our knowledge, this is the first systematic mapping study on MBT approaches using UML ADs. We formulate nine research questions based on the process of MBT, outline our selection criteria, and develop a classification scheme. The results comprise 41 primary studies analyzed against nine research questions. We also highlight the current trends and research gaps in MBT using UML ADs and discuss some shortcomings for researchers and practitioners working in this area.

The remainder of the paper is organized as follows. In Section 2, we present some background knowledge and basic concepts of UML ADs. Section 3 describes the overall MBT process. In Section 4, we outline the protocol of our SMS and the main steps of the process. Section 5 presents the results of the study along with our analysis. In Section 6, we discuss some threats to the validity of this work. Lastly, Section 7 presents our conclusions and some opportunities for future research.

2. UML activity diagrams

The UML Activity Diagram (AD) is an important diagram for modeling the dynamic aspects of a system [10]. Following the Petri nets semantics, the UML ADs use Petri nets concepts such as places, tokens, and control flows [19]. However, the UML AD specification is semi-formal. UML ADs can depict activities (sequential and concurrent), the data objects consumed or produced by them, and the execution order of different actions. Edges are used to control the execution flow of the nodes in an activity. A node does not begin its execution until it receives the control or input on each of its input flows. As a

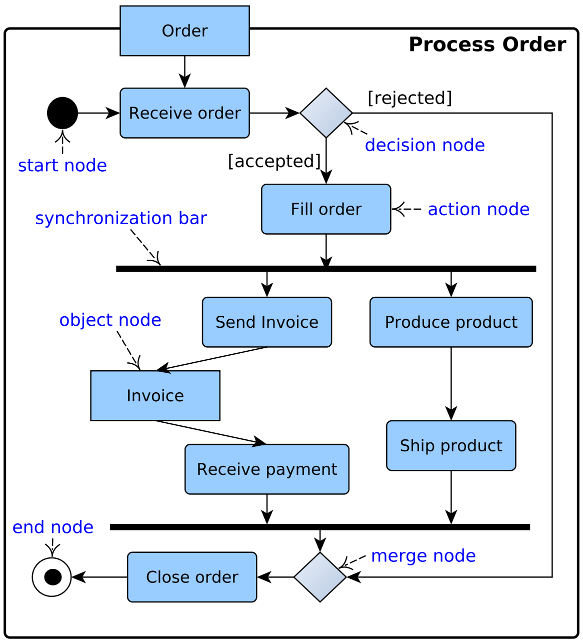

Figure 1: A UML activity diagram with an input parameter [20]

node completes its computation, the execution control transits to the nodes existing on its output flows. The execution of an AD is completed if it reaches a final node and/or returns a data object as a result of the internal computations. Passing parameters to an AD as *Data objects* is possible and used for the exchange of information between two activities and also between two actions in AD. An *action* specifies a single step within an activity. Action nodes are illustrated as round-cornered rectangles while data objects are represented as rectangles. *Edges* are used to control the execution flow of the nodes in an activity. A node waits to execute its computation until it receives the control (or input on each of its input flows), then it begins its execution. As a node completes its computation, the execution control transits to the nodes existing on its output flows. A *Decision* node is shaped like a diamond with one incoming edge and at least two outgoing edges. Each outgoing edge of a decision node is associated with a boolean *condition* such that the edge is chosen if and only if the associated condition holds. Concurrent execution of activities is modeled using *join* and *fork* elements, which are portrayed by synchronization bars where multiple arrows enter or leave, respectively. The *merge* node is typically used to merge paths from join and decision nodes.

For example, the UML AD in Figure 1 specifies the following behavior: The *Process Order* activity starts by receiving an order as an input parameter from a customer at the *Receive order* action. Then, at the diamond-shape branch node, we evaluate the condition with the square brackets on the first branch. If the condition is *true*, we choose that branch and move to *fill order* node, otherwise we reject the order and move to merge node. Once the *fill order* action is executed, we reach at a *fork* (i.e., the first synchronization bar), which splits the path of control flow in two concurrent paths whereas on the right path, we produce and ship the product to the customer, while on the left path, the invoice is sent to the customer and we receive and check the payment against the corresponding *invoice* data object. When both paths have been executed completely, we arrive at the *join* (the second synchronization bar), the control flow moves to merge node. At the end, the *close order* action is executed.

3. Model-based testing

Model-Based Testing (MBT) is a black-box testing technique that generates tests from abstract behavioral models [21]. It allows to automate or semi-automate the entire testing process. Moreover, the testing activities can be left-shifted and the SUT can be tested at an early stage in the software development life cycle. MBT has many benefits including reduced cost, reduced time, and improved test quality [5]. MBT techniques also allow to find defects in requirement specifications and to trace requirements to test cases.

The research objectives for this SMS are based on the process of MBT (see Figure 2). In Section 4.1, we map each step in the MBT process to a research question. The first step in MBT is to create test models of the SUT from the requirements or existing specification documents or to reuse the design models of the SUT. In many cases, the test models are built according to the test objectives and capture only those aspects of the SUT which are relevant for testing. Before using the test models for test case generation, they are validated against the requirements to check that the models correctly capture relevant characteristics and properties of the SUT. Model validation is an important activity but it requires additional efforts [22]. The next step in MBT is to use the test models for test case generation. Test cases are executed against the SUT through test data. In an automated MBT environment, test data is derived from the test models according to the given testing criteria and the input domain of the SUT [1].

The number of generated test cases can be very large due to the high complexity of the models or large input domain. Owing to restricted time and resources, executing the test cases for all possible inputs in all possible states of the SUT is usually not feasible in realistic systems. Thus, the testers use different *test coverage criteria* to decide when to stop the testing process [23], for example, *all-path coverage* (i.e., exercising each execution path at least once) and *branch coverage* (i.e., traversing each possible branch from every decision point in a program under test). *Test coverage metrics* are instruments to measure up to what extent a given set of test cases covers a program or its specifications. These metrics can also be used to measure the *quality* of test cases that are generated from models [24].

The resulting tests are initially on the same level of abstraction as the model focusing only on the test

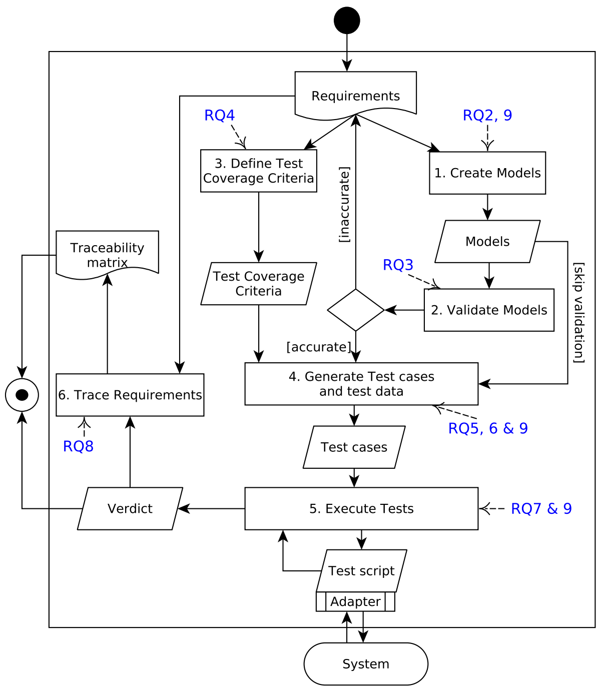

Figure 2: Generic model-based testing process

inputs and outputs. In order to execute these tests against the SUT, they need to be concretized by including test interface specific details. In MBT, there are two techniques to execute the test cases: *offline* and *online*. In offline testing, the generated test cases are written as *test scripts* which can be executed using different test automation frameworks. The benefit of using the offline testing is that the entire test suite can be optimized before execution and stored for later use.

In contrast, in online testing, test generation and test execution are combined such that one test input is generated and test output is received and evaluated and then based on the test output the next test input is generated. In this case the conversion from abstract tests to concrete executable tests is performed by a *test adapter*. As a result, the testers cannot apply post-optimization techniques to the test suite. On the other hand, online testing is adaptive and more suitable than offline testing when the model is non-deterministic.

Generally, the last step of MBT process is to analyze the verdicts of the executed test cases and assemble a *test report* based on the test results and requirements. In addition, in most cases a *traceability matrix* is used to show how different test requirements have been covered by different tests. There are several benefits of a traceability matrix, for example, one can observe which requirements are not being currently covered by test cases and which requirements have been validated.

4. Study design

A systematic mapping study (SMS) is a secondary study that classifies the existing research work related to specific research questions and outlines gaps in the current research [13, 18]. Hence, it can be used as a foundation for new research activities.

We leveraged the well-known guidelines proposed by Petersen et al. [12] for a SMS to conduct our study. The protocol used in this study is the following: based on the research questions (listed in Section 4.1), we specify the search string to collect relevant primary studies[1] from several electronic databases. After analyzing the abstracts of the collected primary studies, we filter out the duplicates and the irrelevant primary studies according to the study selection criteria specified in Section 4.4. Next, we read the full-text of all the remaining primary studies and perform *snowballing* [25] to include further relevant primary studies which were not found during the initial searches to the databases. Based on our quality assessment criteria defined in Section 4.5 we exclude studies which do not meet the minimum quality requirements. Lastly, we extract the required information from the remaining primary studies to answer the research questions. The number of included and excluded primary studies at each stage is shown in Figure 3.

4.1. Research questions

This study aims to identify research gaps and outstanding challenges and suggest where future research is required by classifying the empirical studies on model-based testing approaches using ADs. We post the following research questions, corresponding to each step of the model-based testing process in Figure 2:

RQ 1. Where and when were primary studies published? The aim is to answer following sub-questions:
- What is the annual number of publications in this field?
- Which publication venues (i.e., conferences, journals) are the main targets of studies in this field?

RQ 2. Which other modeling notations have been used in combination with ADs? The aim is to identify if and how other modeling notations have been used to complement the test generation process.

RQ 3. What methods are used for model validation and verification? Although UML ADs 2.x adopt semantics similar to the Petri nets, the specification of the model is still semi-formal. Therefore, unlike Petri nets, UML ADs cannot be formally verified. Answering this RQ will allow us to identify the possible techniques that can be used to validate and verify ADs.

[1]A *primary study* is an empirical study investigating a specific research question [18].

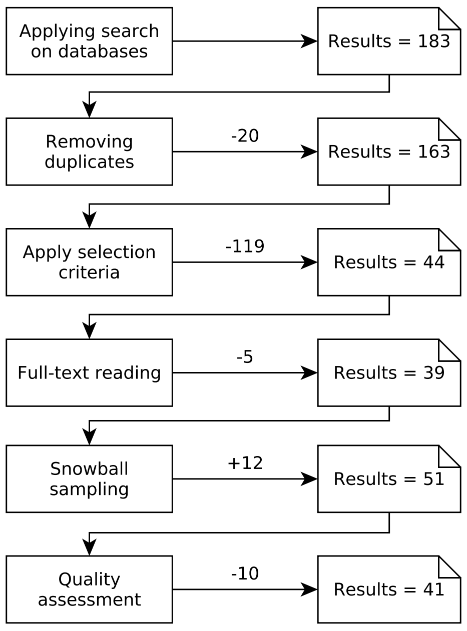

Figure 3: Number of primary studies at each stage during the selection process

RQ 4. What coverage criteria are used for test generation? The objective is to discover the coverage criteria that research has focused upon, to guide the test generation process.

RQ 5. What methods are used for test case generation?

RQ 6. What methods are used for test data generation? Answering this and previous RQ will allow us to gain knowledge about the types of methods for test data and test cases generation, respectively, that have been more prominent.

RQ 7. How are the tests executed against the SUT? The aim is to determine whether online or offline test execution technique has been more popular.

RQ 8. How are the test requirements traced against the model? The aim is to determine how the research community assesses the importance of traceability between the test requirements and the models.

RQ 9. What tools are used for model editing, test generation, and test execution? Answering this RQ will enable us to determine which tools in each of aforementioned categories have been widely adopted by the research community.

4.2. Source selection

In order to get a broader perspective, we searched systematically in electronic databases instead of targeting a constricted set of journals and conference proceedings. Five electronic databases were considered for conducting the searches: *IEEE Xplore, ACM Digital Library, Science Direct, Springer*, and *Web of Science*. These databases contain almost all the important conferences, workshops, and journal publications relevant to the software engineering field [26].

4.3. Search string

Based on our research topic, we used the following search string to collect primary studies:

(Generat*) *AND* (Test *OR* Tests *OR* Test?case *OR* Test?cases *OR* Test?suite) *AND* (Activity?diagram)

The terms were combined using boolean *AND* and *OR* operators in the final search query. The following database fields were used for searching primary studies: *title*, *abstract*, and *keywords*. We used wildcard operators to accommodate different variations of the terms, for instance, we used *generat** for *generate*, *generating*, and *generation*. The wildcard operator *?* is used to match only one character. Table 1 lists the number of primary studies found from each database.

Table 1: Number of primary studies per database

Database	**Search results**
IEEE	53
ACM	17
Web of Science	39
Springer	62
Science Direct	12
Total	**183**

4.4. Study selection criteria

Our objective was to consider only peer-reviewed and published research publications that contain sufficient technical details. In order to select the most relevant primary studies for the SMS, three researchers undertook the study selection. After executing our search queries on all selected electronic databases, each researcher filtered the search results independently and reviewed the results at the end of the study selection process. We also used *snowballing* to complement the electronic search [12].

In case of differences in the selection results among researchers or when a researcher could not decide whether to keep or exclude a study, that study was discussed with the other researchers and a decision was made. In our selection process, we applied the following inclusion criteria:

- Papers whose *abstracts*, *titles* or *keywords* discussed test case generation using activity diagram or any of the alternate terms that we specified in Section 4.3;
- Papers written in English;
- If an extended version (e.g., book chapter or journal paper) of a conference paper was found in the search results with more technical details, only the extended version was included.

The following were the exclusion criteria used to discard irrelevant studies:

- The publication is a secondary study (e.g., a literature review);
- Papers not subject to peer review;
- Duplicated papers (e.g., returned by different search engines).

After applying the inclusion and exclusion criteria, 119 studies were excluded. During full-text analysis, another five studies were excluded as they were not in the scope based on the selection criteria. The remaining papers (i.e., 39) were used to conduct backward snowball sampling [25, 12] (as shown in Figure 3), which led to 12 primary studies being added.

4.5. Study quality assessment

We designed a questionnaire for quality assessment (as suggested in [26]) in order to evaluate the quality of the selected primary studies. The quality criteria include the following questions:

1. Are the goals clearly described?
2. Is the method/algorithm clearly described?

3. Are assumptions/restrictions clearly described?
4. Is the method validated via a case study?
5. Is tool support discussed?
6. Is the case study realistic?
7. Are there multiple case studies used for validation?
8. Is there a qualitative comparison with other approaches?
9. Is there a quantitative comparison with other approaches?

Each primary study was evaluated based on the questionnaire mentioned above, in which each question was scored as follows [27, 28]:

- Fully answers the question: 2 points;
- Partially answers the question: 1 point;
- Does not answer the question: 0 points.

A primary study with a high score is considered to be more relevant to and contributing more to the overall conclusion of our study. A study can have maximum 18 points and minimum 0 points. We defined 4.5 points as our cut-off point (i.e., first quartile) of our quality assessment scale [27, 28]. The studies with a score of 4.5 or less were excluded from this study. As shown in Figure 3, a total of 41 primary studies were finally included.

4.6. Data Extraction

In order to answer the research questions, we designed a data extraction form shown in Table 2 to aggregate data from all the primary studies. We gathered two types of information from each study: *general* information like paper title, authors and publication year; and information directly related to the research questions.

In order to avoid errors and to mitigate researcher's biasedness during the data extraction phase, two researchers extracted the data from the selected papers while two other researchers validated the extracted data. All discrepancies and disagreements were resolved in consensus meetings and discussions.

Table 2: Data extraction form

Type	**Data item**	**Type of information collected**
General	Paper title	Title of the primary study
	Authors	Set of names of the authors
	Year	Calendar year of publication
RQ1	Publication venue	Name of publication venue
RQ2	Modeling notations	Modeling notations used for test generation
RQ3	Validation techniques	Techniques/methods used for model validation
RQ4	Coverage criteria	Coverage criteria used to guide test generation
RQ5	Test generation	Types of methods used for test case generation using ADs
RQ6	Test data generation	Types of method used for test data generation using ADs
RQ7	Test execution	Types of test execution techniques
RQ8	Traceability	Approaches to trace test requirements against ADs
RQ9	Tool set	Tools used for model editing, test generation, and test execution

For validating our results, we created a word cloud of the most frequent words occurring in the titles and abstracts of the selected primary studies (see Figure 4). We eliminated the common English words such as *this*, *first*, etc. Further, we grouped different variations of the same word, for instance, *generating, generation*, and *generated* as *generate*. As seen in Figure 4, the top 5 most frequent words were *test, generate, case, activity*, and *diagram*, which correspond to our selection criteria.

4.7. Schedule of the study

Table 3 shows the detailed schedule of our study, listing all the main steps we performed to compile this study. The steps in this mapping process may appear to be sequential, but in our experience, some of them were iterative. For instance, the data extraction phase matures and updates through the process since new information is observed while reading through the selected primary studies, which demands more time and efforts than anticipated. Furthermore, we experienced that presenting and organizing the mapping results comprehensively and coherently was a considerably time-consuming and challenging task as well. It should be noted that the search for primary studies was completed on December 30, 2016. Therefore, any papers published after December 30, 2016 are not included in the results.

Table 3: Schedule of the study

No.	**Step**	**Duration (business days)**	**Due date**
1	Study planning	2	13.12.2016
2	Formulate protocol for the study	5	23.12.2016
3	Conduct database searches	5	30.12.2016
4	Filter studies based on title and abstract	7	13.01.2017
5	Filter studies based on full-text	8	26.01.2017
6	Snowball sampling	7	03.02.2017
7	Filter additional studies from snowballing	3	09.02.2017
8	Extract and process data	15	03.03.2017
9	Perform quality assessment	3	09.03.2017
10	Prepare first draft of the paper	15	03.04.2017

5. Results and analysis

In the following, we analyze and discuss the results for each of the nine research questions listed in Section 4.1.

5.1. Frequency and venues of publication (RQ1)

In this study, peer-reviewed venues (journals, conferences, and workshops) were considered. The bar chart in Figure 5 illustrates the distribution of the primary studies over years and publication types. Most of the selected primary studies (i.e., 28 out of 41) were published as conference papers. The top three venues based on the number of publications are listed in Table 4.

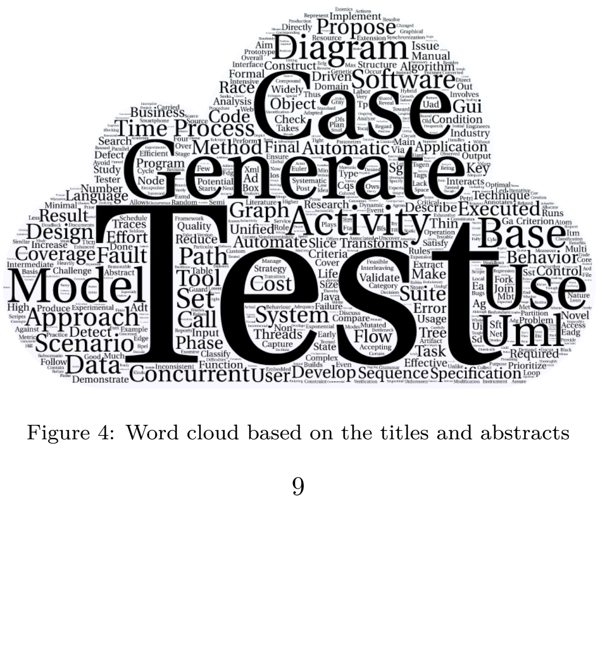

Figure 4: Word cloud based on the titles and abstracts

Table 4: Top publication venues

Rank	Venue	Reference	Count
1	Asia-Pacific Software Engineering Conference	[2, 29, 30, 31]	4
2	International Computer Software and Applications Conference	[32, 33, 34]	3
2	SIGSOFT Software Engineering Notes	[35, 36, 37]	3

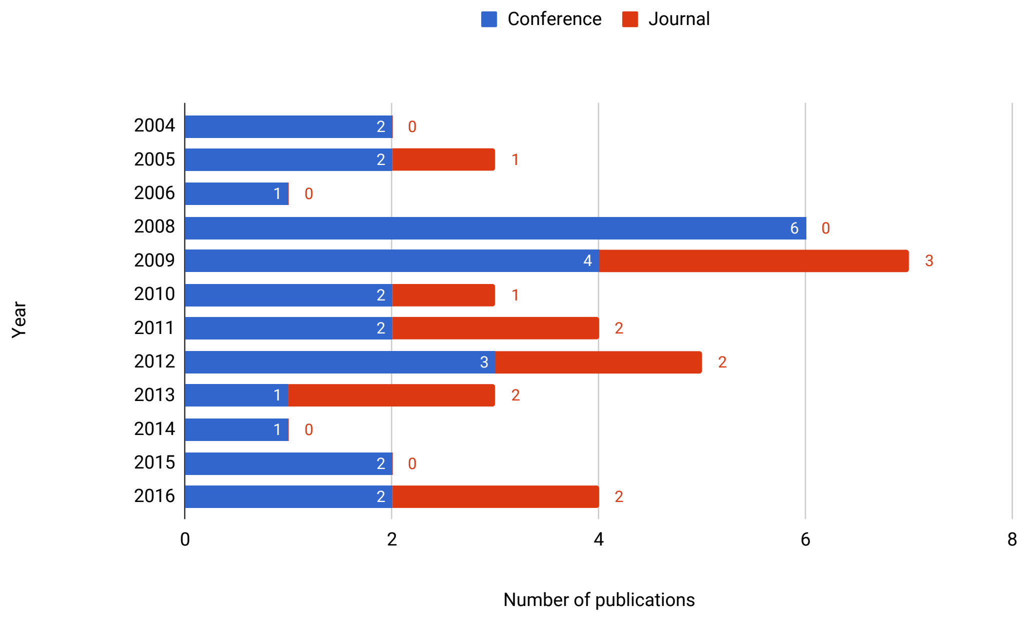

Figure 5: Publication distribution over years and publication types

5.2. *Modeling notations (RQ2)*

We have categorized the studies based on the modeling notations that have been used for test case generation, as listed in Table 5. The results show that in most of the studies (i.e., 29 out of 41), ADs are transformed into other formalisms before performing test cases generation. Only in 12 studies, tests are generated directly from ADs (see Figure 6).

In most of the cases in the first category [29, 30, 39, 40, 41, 42, 43, 45], ADs are translated into *directed graphs* where each vertex denotes a node in the AD and each transition from one node in the AD to another node is specified by an edge in the graph. One of the main benefits of transforming an AD into an activity graph is that one can use several well-established graph-search algorithms for generating test cases with respect to given graph-based coverage criteria. However, Hettab *et al.* [38] claimed that the activity graphs do not encapsulate all the features of an AD which are necessary for test generation and proposed an *Extended Activity Dependency Graph* (EADG). A vertex in an EADG can represent a set of nodes in the AD, instead of a single node, implying that the nodes inside the vertex are concurrent and they can be executed simultaneously. Furthermore, Shirole *et al.* [44] and Nejad *et al.* [46] transformed ADs into *Control Flow Graphs* (CFGs) which are structurally similar to activity graphs. Samuel *et al.* [35] constructed a *Flow Dependence Graph* (FDG) from an AD, in order to represent the execution flow and data dependencies in the AD. Heinecke *et al.* [11] presented a method for generating test plans from ADs. In their proposed approach, an AD is translated into an *Interaction Flow Diagram* (IFD). The IFD illustrates only the control flow of inputs and outputs, a tester has to provide to and expect from the SUT, respectively. The internal actions (or activities) which are transparent for the tester are not included in the IFD. The purpose of the

Table 5: Classification of studies according to the modeling notations used

Notation	**Reference**	**Count**
Graphs	[32, 29, 30, 38, 39, 35, 11, 40, 41, 42, 43, 44, 45, 46, 47, 48]	16
Activity diagram	[49, 2, 50, 51, 52, 53, 54, 55, 34, 56, 57, 37]	12
Use-case / Sequence / Class diagram	[58, 36, 59, 37, 60]	5
Trees	[33, 61, 31, 62]	4
Executable test specification	[63, 64, 58]	3
Formal models	[65, 66]	2

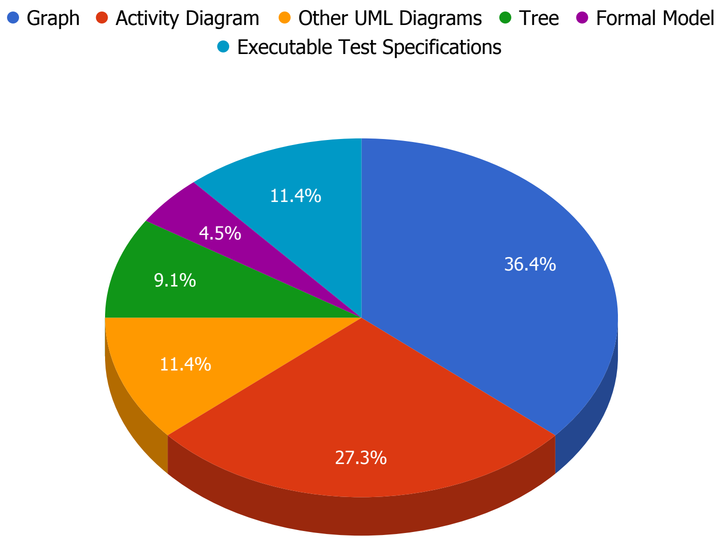

Figure 6: Distribution of studies according to modeling notations used

IFD is to serve as an initial point to develop a test plan. Li *et al.* [47] transformed an AD to a directed graph and, later on, to extract Euler circuit for test case generation.

Nayak and Samanta [48] claimed that it is difficult to interpret the execution orders of the activities in ADs due to complex dependencies that arise within nested structures. They proposed an approach to transform an AD into a well-formed hierarchical structure, known as *Intermediate Testable Model* (ITM), which indicates high-level flows of control in the AD. The main purpose of the ITM is to understand the semantics of the control flow of an AD unambiguously and use it for generating test scenarios.

In almost 30% of the studies, ADs are directly used for generating test cases. Lie *et al.* [56] and Shirole and Kumar [37] have extended the Activity Diagrams and Sequence Diagrams (SD) by associating the data access tags to the activities and operations in order to test the data race conditions and inconsistencies of the program under test (PUT). Similarly, Gantait [57] marked the activity nodes in an AD with different stereotypes to identify which activities require input data from the user and which activities show the expected output in a given test case.

In some studies, other UML diagrams such as *Use-case*, *Sequence* and *Class Diagrams* were also employed to complement ADs in the test case generation process. Tiwari and Gupta [59] and Akour *et al.* [60] transformed use-case diagrams into ADs for test case generation. Similarly, Shirole and Kumar [36] proposed an approach to convert sequence diagrams into ADs for test generation.

We found 4 studies where ADs were converted into trees. Sun [33] and Sun *et al.* [62] have stated several rules to transform an AD to a set of *Extended AND-OR Binary Trees* (EBT). The fork and join nodes are replaced during the transformation with concurrent structures represented as binary trees to prevent the path explosion problem during the test case generation. The EBTs are used for generating test scenarios with respect to a coverage criterion for concurrent flows. Tiwari and Gupta [31] specified several mapping rules to transform an AD into a *Software Success Tree* (SST) and a *Software Fault Tree* (SFT). These trees are used to build the minimum cut sets that are used to generate test cases for testing the normal and exceptional behavior of the SUT. Kansomkeat *et al.* [61] constructed *Condition-Classification Trees* by analyzing decision points and guard conditions in an AD. These trees are used to generate a test case table and then test cases.

Hartmann *et al.* [63] and Vieira *et al.* [64] translated an AD into the *test specification language* (TSL) format which is an input language of Test Development Environment (TDE) [67] tool for test generation. The TSL test design is produced by mapping the activities and transitions of the AD to TSL partitions and choices, respectively. Yuan *et al.* [58] used UML ADs for modeling business processes under test. They transformed the ADs representing business processes into abstract test cases (visualized as UML sequence diagrams) and then the abstract test cases into executable test scripts in TTCN-3 (Testing and Test Control Notation version 3). Since the study uses ADs with sequence diagrams and executable test specifications, [58] has two entries in Table 5.

Farooq *et al.* [66] transformed an AD into a *Colored Petri-nets* (CPN) model by following the transformation rules specified in [68]. Similarly, Chen *et al.* [65] proposed an approach to convert a UML AD into a *Symbolic Model Verifier* (SMV) model [69]. Unlike ADs, the CPN and SMV models are formal.

5.3. Model validation and verification (RQ3)

We have found three studies in which ADs are translated into formal models. Chen *et al.* [65] specified several translation rules to transform ADs into SMV models. Similarly, Farooq *et al.* [66] derived a CPN model from a given AD model. The CPN model is used to represent the control flow of the AD model. Tiwari and Gupta [31] converted an AD into a SFT. The resulted tree can be used as an input to OpenFTA [70] tool for structural validation. These models can be employed to verify the correctness of the specifications and guarantee the quality of the models. However, these approaches do not explicitly verify the derived formal models. Furthermore, we have observed that none of the studies syntactically validate ADs at the design level, for instance, using the Object Constraint Language (OCL) [71].

5.4. Coverage criteria used for test case generation (RQ4)

Table 6 presents a list of coverage criteria identified during data extraction. We have categorised these coverage criteria concerning graph, logic, and data coverage. The results show that coverage criteria related to the graph are more prevalent among studies. Approximately 71 % (32) of the studies use graph-based coverage criteria which include path, node (action, state) and edges (transition). The data related criteria are the second most popular coverage criteria in this study where approximately 7 % (3) studies used it. Among the least popular, logic coverage is discussed only in less than 5% (2) of the studies. Moreover, in more than 17% (8) of the studies, no coverage criteria were explicitly mentioned. However, 35 % (15) of the studies use a combination of several coverage criteria as shown in Table 7.

Table 6: Types of test coverage criteria used in different studies

Category	Criteria	Description	References	Count
	(All) Basic path coverage	Each activity in the path visited only once.	[2, 40, 43]	3
	Simple path coverage	Each edge in the path to be visited only once.	[49, 50, 55]	3

Continued on next page

Graph

Category	Criteria	Description	References	Count
	Happy path coverage	Cover the default path without exceptions or error conditions	[64]	1
	Basic path for non-concurrent activities	Each activity occurs at most once. For loop structure, each activity occurs at most twice.	[38]	1
	(All) Path coverage criteria	All possible path to be covered (at least once).	[50, 35, 11, 54, 59, 39, 45, 41, 42, 61]	10
	Representative path coverage	One of the prime paths in activity diagram to be covered.	[34]	1
	Key path coverage	A path is a key path if there is no state repetition in path.	[53, 65]	2
	Simple path for concurrent activities	In a basic activity diagram, if there exists many basic paths that have the same set of activities and the same partial order relation, we only select one representative from this path set. The selected representative is called a simple path of the activity diagram.	[38]	1
	Activity/State/ Node coverage	Requires all activities states in activity diagram to be covered.	[50, 72, 53, 65, 47, 59, 49, 58]	8
	Action coverage	All the actions to be covered.	[55]	1
	Transition/Edge coverage	Requires all transitions/edges in activity diagram be covered.	[50, 72, 53, 65, 57, 47, 55, 66, 63, 49, 58]	11
	Branch coverage	For decision node in an AD, any/all outgoing edge to be coverage, respectively, for branch and full branch coverage.	[51, 40, 66, 72]	4
	Selection coverage criterion	For each decision node, all the outgoing edges to be covered.	[48]	1
	Interaction coverage	All the interactions in an AD to be covered (it is a super set of activity and transition coverage).	[65]	1
	Loop adequacy	For each pre-test loop node, the loop to be taken: (1) zero time (body of the loop must be skipped) (2) at least once.	[48]	1
	Concurrency coverage criteria	To cover one/all feasible sequences of activities without/with interleaving activities between parallel processes.	[52, 62, 33]	3
	Concurrent (Path) coverage	For each concurrent node, every valid interleaving of actions to be taken.	[48, 44]	2
	Interleaving node/edge coverage	The execution of n-wise permutated set of concurrent nodes/edges in n synchronized sub processes.	[66]	1
	Join and fork state coverage	Each edge from a fork node leading to a join node to be covered	[72]	1
Logic	(Full) Predicate coverage criteria	Each predicate on any edge must be tested for True and False values, respectively.	[40, 35]	2

Continued on next page

Table 7: Multiple coverage criteria used by different primary studies

Reference	Coverage criteria combination
Chen *et al.* [49]	Simple path, activity/state/node, transition/edge
Chen *et al.* [65]	Key path, interaction, activity/node/edge
Vieira *et al.* [64]	Happy path, data
Hettab *et al.* [38]	Basic path for non-concurrent activities, simple path for concurrent activities
Yuan *et al.* [58]	Activity/state/node, transition/edge
Sapna *et al.* [50]	Simple and all paths, activity/state/node, transition/edge
Farooq *et al.* [66]	Interleaving node/edge, transition/edge, branch
Samuel and Mall [35]	All dependency path, predicate, boundary testing
Fan *et al.* [72]	Activity/state/node, transition/edge, join and fork, branch
Sapna *et al.* [53]	Key path, activity/state/node, transition/edge
Boghdady *et. at.* [40]	Basic path, branch, predicate
Nayak *et al.* [48]	Selection, loop adequacy, concurrent path
Li. *et al.* [47]	Activity/state/node, transition/edge
Li. *et al.* [55]	Simple path, action, transition/edge
Tiwari *et al.* [59]	All path, activity/state/node

Category	Criteria	Description	References	Count
Data	Data coverage	Include sampling, group coverage expression, choice coverage and exhaustive coverage	[64]	1
	Boundary value testing criterion	Each predicate excluding equality to be covered via providing the extremes and the corner values of the input domain boundaries.	[35]	1
	CUT-Set coverage	To cover all CUT-set (a set of minimum inputs leading to the top event) in a Success Tree.	[31]	1
Not specified			[32, 56, 36, 37, 46, 60, 29, 30]	8

5.5. Test case generation methods (RQ5)

We have grouped the approaches based on the methods that have been used for test case generation, as listed in Table 8. As discussed in Section 5.2, in most of the studies, ADs are transformed into trees or graphs. These intermediate structures are later used for generating test cases using different graph search methods. In most cases [29, 38, 58, 33, 53, 40, 48, 57, 43, 45, 55, 62, 60], *Depth First Search* (DFS) has been used for test case generation. However, in some studies [39, 41, 59, 37], DFS has been used in combination with *Breath First Search* (BFS). BFS is used to traverse the concurrent vertices while the DFS is employed to traverse the rest of the vertices of the graph. Furthermore, in [2, 11, 56], the DFS algorithm has been modified in such a way that it visits each cycle only once. Fan *et al.* [72] proposed an approach for generating test cases from *Compound Activity Diagram* (CAD). CAD is a high-level UML AD which consists of subactivity diagrams. They used the DFS to generate test cases for the subactivity diagrams and combined the obtained test cases using the Round-Robin method.

Li *et al.* [47] presented an approach to build the Euler circuit from an AD and then generate test cases using Euler circuit algorithm. They first transformed the AD to a directed graph and then used the graph to construct an Euler circuit. A *circuit* is a path in a graph which starts and ends at the same vertex. An Euler circuit is a circuit of a graph if it exactly traverses each edge in the graph only once. Therefore, an Euler circuit can be used to satisfy the transition coverage criteria with the minimum number of test cases.

Samuel and Mall [35] proposed a method of test case generation using dynamic slicing. *Program slicing* is a decomposition technique which extracts only those program statements that are relevant to a particular computation. Dynamic slicing investigates just a specific execution of the program to compute the program slice by gathering the run-time information. They used FDG as an intermediate representation of an AD to derive dynamic slices. In their proposed approach, the slices are built corresponding to each conditional predicate on the activity edges. Test cases are generated with respect to each slice. Shirole and Kumar [36] generated test sequences for concurrency testing using the *Concurrent Queue Search* (CQS) algorithm. This algorithm constructs a queue for each thread of execution. The concurrent queues are generated from the AD and later combined and dequeued according to the control flow of the AD to form a test path. The proposed CQS method is designed to reveal the data safety errors in the presence of concurrency. Similarly, Sapna and Mohanty [50] proposed an approach to reduce the number of generated test scenarios from an AD when considering concurrent activities. The proposed approach generates only valid test scenarios by taking into account the data dependencies between the concurrent activities in the AD.

Tiwari and Gupta [31] translated an AD into a SST and a SFT. These trees are analyzed to produce the minimum cut sets specifying a minimum number of conditions that need to be satisfied in order to force the SUT into a certain state. The cut sets obtained from SST and SFT are used to generate test cases for testing the normal and exceptional behavior of the SUT, respectively. In another approach presented by Xiaoqing *et al.* [32], an AD is converted into a *thin-thread tree* by finding all the execution paths from the AD. Each execution path from the root to a leaf node in that tree represents a test case. Sun *et al.* [52] proposed an approach to decompose an AD to several modules which are based on transitions, and then generate test sequences for each module and store them in separate files. Based on the given coverage criteria, the test sequences for different modules stored in separate files, are later combined to form the final test cases.

Chen *et al.* [49], Farooq and Lam [51], and Farooq *et al.* [66] randomly generated test cases and selected test cases according to given coverage criteria. For instance, Chen *et al.* [49] instrumented a Java program under test and generated a set of test cases randomly. Next, the execution traces of the instrumented program were collected by running the program against the generated test cases. These traces were compared with the AD to produce a reduced set of test cases according to given test adequacy criteria. Farooq and Lam [51] optimized test suites using the *Multi-objective evolutionary algorithms* (MOEA). These test suites are generated using the random walk based algorithm. Similarly, Farooq *et al.* [66] transformed an AD into a CPN model and used the random walk based algorithm on the CPN model to generate test cases. Chen *et al.* [65] introduced a method for generating test cases using several model-checking techniques. They proposed coverage-driven mapping rules to transform an AD to the input specification of the Cadence SMV [73] model checker. Similarly, Sun *et al.* [34] presented a test case generation approach based on mutated ADs using the Yices Satisfiability Modulo Theories (SMT) Solver [74]. To this extent, test cases are obtained by solving the mutated activity path constrains, which include conditions for weakly killing the relative mutants.

Shirole *et al.* [44] and Nejad *et al.* [46] applied *Genetic Algorithms* (GAs) directly to activity graphs

Table 8: Test case generation methods

Method	**References**	**Count**
Graph algorithms	[2, 29, 38, 58, 33, 56, 39, 72, 53, 11, 40, 48, 57, 41, 43, 45, 55, 59, 37, 62, 50, 35, 52, 36, 31, 47, 32, 60]	28
Random search	[49, 66, 51]	3
Model checking	[65, 34]	2
Heuristic algorithms	[42, 44, 45, 46]	4
Condition-classification tree method (CCTM)	[61]	1
Adaptive agents	[30]	1
Category-partition method	[63, 64]	2
Not specified	[54]	1

in order to generate test cases which satisfy given coverage criteria. Sumalatha and Raju [42] proposed a technique to generate all the unique paths in the activity graph (derived from an AD) by covering all the edges and then using a GA to obtain the best test cases. Similarly, Mahali and Acharya [45] proposed an approach to generate all test paths using DFS and then to employ a GA to prioritize them. Since they used both a graph algorithm and a heuristic, [45] has two entries in Table 8. Kansomkeat *et al.* [61] proposed a *Condition-Classification Tree Method* (CCTM) for generating test cases from ADs. Classifications are criteria for partitioning the input domain of the program under test, and classes are the disjoint subsets of values for each classification. A classification tree arranges the classifications and classes into a hierarchical structure. Condition-classification trees are derived from the ADs by analyzing decision points and guard conditions in the diagrams. The trees are then used to create test case tables and test cases.

Xu *et al.* [30] introduced a technique to generate test scenarios from an AD using the *adaptive agents*, which effectively explore the AD. Hartman *et al.* [63] and Vieira *et al.* [64] used a Test Development Environment (TDE) [67] tool for generating test cases. The TDE tool is based on the category-partition method, which determines behavioural equivalence classes within the structure of SUT. Lastly, Ye *et al.* [54] presented an approach to build a test suite for regression testing. The test suite consists of reusable test cases from the existing pool of test cases that cover the affected paths and newly generated test cases which cover the new paths in the AD. However, they did not explicitly specify the method used for test case generation.

5.6. Test data generation methods (RQ6)

Test data generation in software testing is the process of identifying a set of test data which satisfies given testing criterion. It may be the actual data that has been taken from previous operations or a new set of data explicitly created for this purpose as discussed by Korel *et al.* [75]. Table 9 presents a distribution of test data generation methods we identified during the data extraction phase. In more than 50% (23) of the selected studies, no test data generation method was explicitly specified. The most popular method among selected studies to generate test data is the category-partition method.

By using the category-partition method, the functional test cases for the major functions are created by decomposing functional specifications. By identifying the elements that influence the functionality, test cases are generated by methodically varying the elements over all the values of interest. The categories are defined as the primary characteristics of the input domain of the function under test. Each category is partitioned into equivalence classes of inputs called choices. By definition, choices in each category must be disjoint, and together the choices in each category must cover the input domain [1].

Table 9 also depicts the other studies which addressed test data generation. We briefly describe these approaches below. Yuan *et al.* [58] used combinatorial logic coverage to generate test data, by considering all possible combinations of values at the boundary, within the boundary, outside of the boundary, and precisely on the boundary of the input parameters. Sapna and Mohanty [50] used the priority based method to select activities on the same level in an activity diagram. In order to generate test data for activities, the priority of an activity is identified based on the dependencies and later used for the test data generation. Activities inside a fork-join can be said to be at different levels of execution wherein activities at one level must be completed before moving to activities at the next level. In order to avoid data race conditions in thin-threads generated from UML ADs, Xiaoqing *et al.* [32] used data-object trees. The data-objects trees along with condition-trees and thin-thread are used for scenario-based testing.

Samuel and Mall [35] used CFGs and the edge marking method to generate dynamic slices from an activity diagram based on the true or false values of the predicates. The test data values are generated subject to the slice conditions. All input values which satisfy a slice condition are mapped to a minimization function which reduces the problem set through repeated modifications of input data values as proposed by Korel [75]. The condition-classification tree method is used by Kansomkeat *et al.* [61] to generate test data. The marked intersections of test case table and condition-classification tree are used to generate test data. Each intersection represents an applicable condition, evaluated either true or false, providing test conditions for the test cases.

Shirole and Kumar [36] proposed a concurrent queue search algorithm to generate test scenarios to handle concurrent tasks. The sequence diagrams are converted to ADs which encapsulate sequential, conditional, iterative, and concurrent flows of the task. The algorithm proposes a queue for each executing thread that is later used to select a switching point to generate a feasible concurrent test sequence based on the test data already present in the model. Tiwari and Gupta [59] created actor-oriented ADs and then generated systematically prioritized test sequences. The use case dependencies with pre- and post-conditions are modelled in activity via priority field to identify the path with high preference and act as test data for the respective test case.

Sun *et al.* [34] proposed mutation operators for UML ADs and used mutation testing approach to generate mutated UML ADs. The proposed mutation operators are applicable to various elements of activity diagram namely action nodes, control nodes, object nodes, flow, and expressions. By applying the operators, the mutated ADs are used to generate paths and constrains that are later used to generate test data. Shirole and Kumar [37] presented an extension of UML models to represent data-access tags which serve as test data and are used to generate data-access traces. The generated data-access traces are utilized by an extended state machine to identify data-race and blocking in concurrency testing. The approach has limited applicability regarding ADs to find data-race and blocking scenarios due to unavailability of sufficient concurrency constructs in ADs to specify all concurrent behaviors. Gantiat [57] adopted a manual approach to generate test data for identifying prioritized test paths. The test data entered as an input determine the selected path in the AD at runtime. For each possible flow in the AD, the predicates are identified and the values to test them are entered manually.

Figure 7 presents the number of primary studies (as bubble size) with respect to the different test case generation methods from Table 8 and the test data generation methods from Table 9. Again, [45] appears twice. The results show that in 14 out of 28 studies involving graph algorithms for test case generation, no test data generation method was explicitly specified. Similarly, in all four studies involving heuristic algorithms for test case generation, no test data generation method was specified. Moreover, six studies in which graph algorithms were used for test case generation the category-partition method was used for test data generation. Additionally, in two studies the category-partition method was used for both test case and test data generation. Similarly, in one study CCTM was used for both purposes.

5.7. Test execution techniques (RQ7)

We have grouped all the primary studies into 4 categories concerning test execution, as listed in Table 10. In nearly 50% (20) of the studies, test cases are executed offline. Only in one study [65], the SMV model checker [73] was used to execute test cases online against the SUT. Moreover, in almost half of the studies (20 studies), no test execution technique was explicitly specified. It should be noted that in most of these studies, the generated test cases are abstract. Therefore, they cannot be executed in an online fashion.

Table 9: Test data generation methods

Method	**References**	**Count**
Combinatorial logic coverage	[58], [31]	2
Priority and level based	[50]	1
Data-object trees	[32]	1
Alternate variable method	[35]	1
Condition-classification tree method (CCTM)	[61]	1
Manually	[57]	1
Use case dependencies	[59]	1
Category-partition method	[2, 64, 33, 62, 56, 48, 43, 63]	8
Mutation	[34]	1
Concurrent access	[37]	1
Not specified	[30, 38, 51, 72, 53, 65, 11, 40, 54, 44, 47, 55, 66, 52, 45, 49, 39, 41, 29, 60, 46, 42, 36]	23

5.8. Approaches to trace test requirements against the models (RQ8)

The only study that mentions traceability of requirements to and from tests is presented by Boghdady *et al.* [40]. They suggested that the IBM Requisite Pro tool can be used to trace requirements without further exploration.

5.9. Tools for model editing, test case generation and test execution (RQ9)

More than half of the primary studies surveyed (22 out of 41) do not explicitly mention any tool for model editing, test case generation, or test execution. The remaining studies specify different tools used to support their presented approaches. We classified them into three categories, namely tools for *model editing*, *test generation*, and *test execution*. Table 11 provides an overview of the identified tools categorized into commercial, open source, free, in-house, and free community edition tools.

As seen in Table 11, most of the primary studies [2, 63, 30, 64, 56, 35, 48, 45, 40, 43, 55, 65, 29] use commercial tools and only a few [64, 52, 11, 29, 62] use open source tools for making and editing ADs. In addition, two studies [40, 43] have reported the use of a free community edition tool.

With respect to test generation, most of the identified tools are developed in-house. For instance, Linzhang *et al.* [2] developed a prototype tool called UMLTGF using the Rational Rose Extensibility Interface. The tool consists of a UML model parser and a test case generator. The parser can import and

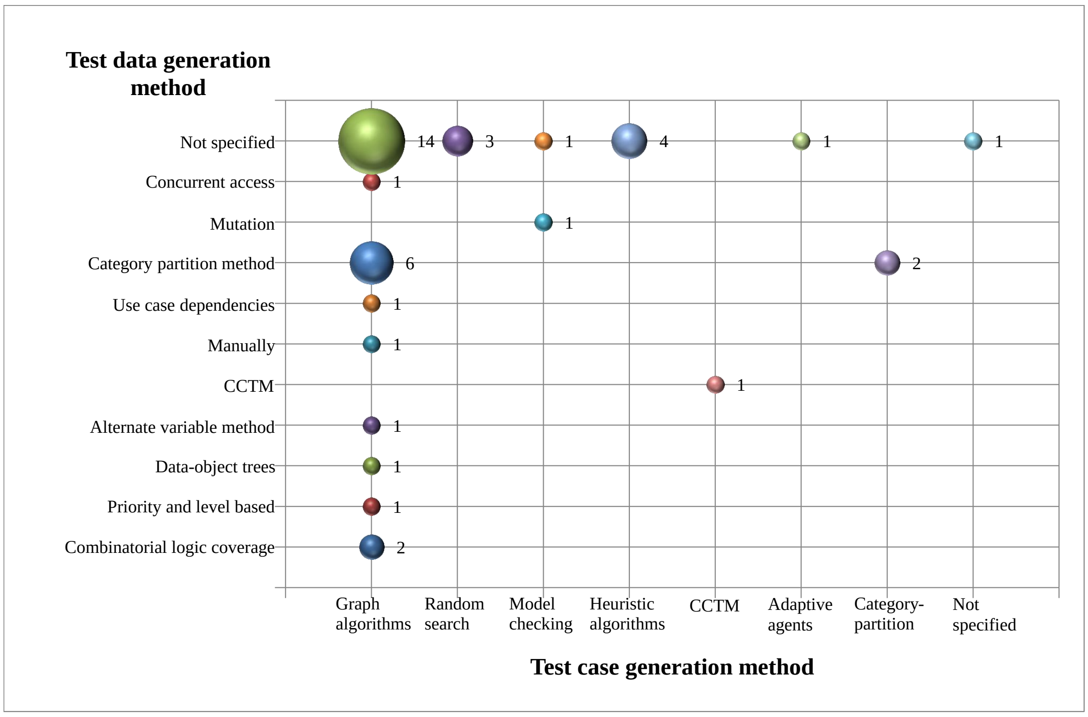

Figure 7: Number of primary studies (as bubble size) with respect to test case generation and test data generation methods

Table 10: Test execution techniques

Execution Method	References	Count
Offline	[2, 11, 49, 63, 30, 38, 58, 33, 56, 66, 35, 53, 61, 57, 43, 54, 44, 55, 62, 34]	20
Online	[65]	1
Not specified	[32, 29, 64, 50, 51, 39, 52, 72, 40, 48, 41, 42, 36, 31, 47, 45, 59, 37, 60, 46]	20

Table 11: Tool support

Type	Tool	Availability	Reference	Count
Model editing	Rational Rose	Commercial	[2, 63, 30, 64]	4
	Poseidon	Commercial	[30, 29]	2
	Visio	Commercial	[30]	1
	Borland Together	Commercial	[64, 56, 65, 29]	4
	Argo UML	Open source	[64, 52, 29, 62]	4
	MagicDraw	Commercial	[35, 48]	2
	Eclipse Modeling Framework (EMF)	Open source	[11]	1
	Smart Draw	Commercial	[45]	1
	Visual Paradigm	Commercial, Free community edition	[40, 43]	2
	Enterprise Architect	Commercial	[40, 55]	2
	IBM Rational Modeler	Commercial	[40]	1
Test generation	UMLTGF using Rose Extensibility Interface	In-house	[2]	1
	TSGAD (Test Scenarios Generator for Activity Diagrams)	In-house	[30]	1
	AGTCG tool	In-house	[49]	1
	AToM^3	Free	[38]	1
	UTG (UML behavioral Test case Generator)	In-house	[35]	1
	Add-on for Rational Rose (TDE/UML)	In-house	[63]	1
	Plugin for Eclipse Modeling Framework	In-house	[11]	1
	Yices SMT Solver	Open source	[34]	1
	Cadnece SMV model checker	Open source (discontinued)	[65]	1
Test execution	Test Complete	Commercial	[40]	1
	IBM Rational Robot	Commercial	[40]	1
	IBM Rational Functional Tester	Commercial	[40]	1
	Automated QA	Commercial	[40]	1
	XCode Instruments	Free	[55]	1
	Android Robotium	Open source	[55]	1
	Compuware TestPartner	Commercial	[63]	1
Not specified	[32, 58, 33, 50, 66, 39, 72, 53, 61, 57, 41, 42, 54, 36, 44, 59, 37, 60, 46, 31, 51, 47]			22

extract the ADs directly from UML models. It also analyses the semantics of the parsed model and derives test scenarios satisfying the proposed coverage criteria. The test case generator analyses each test scenario and generates test cases which can be reused, modified and stored by using a test case manager. Dong Xu *et al.* [30] developed a prototype tool called TSGAD (Test Scenarios Generator for Activity Diagrams) using their proposed algorithm to automatically generate test scenarios for ADs developed using any standard UML tool and exported into a XMI file. TSGAD can directly read an XMI file. AGTCG is developed by Mingsong *et al.* [49] to instrument a Java program according to a given AD. It uses randomly generated test cases to run the instrumented program and gather the corresponding program execution traces. By comparing the collected traces with the behavior of the AD, the tool can group test cases into a test suite according to a given test adequacy criterion. Furthermore, it evaluates to which extent the test suite satisfies the test adequacy criterion. The tool can also be used to check the consistency between the program execution traces and the behavior specified by ADs.

A Java-based prototype tool UTG (UML behavioural Test case Generator) is implemented by Samuel and Mall [35]. The tool is capable of integrating with other UML CASE tools which support exporting and importing models in XML format. Since UTG takes UML models in XML format as input, UTG is independent of any specific CASE tool. The tool uses an FDG to generate test cases based on the proposed slicing algorithm, and later on, each slice is used for test data generation. AToM^3 (A Tool for Multi-formalism Meta-Modelling) [38] is a free tool for multi-paradigm modeling, written in the Python programming language. The formalisms and models are described as graphs. It uses a modified entity-relationship diagram to create meta models that are transformed into graph-grammar models via graph rewriting rules. The proposed tool takes an EADG as input and generates a set of test cases for both concurrent and non-concurrent activities.

Hartmann *et al.* [63] used a Rational Rose plug-in called TDE, a product developed by Siemens corporate research. TDE processes a test design written in the test-specification language (TSL). This language is based on the category-partition method, which identifies behavioral equivalence classes within the structure of a system under test. A recursive, directed graph is built by the TDE that has a root category/partition and contains all different paths of choices to plain data choices and creates test cases to satisfy all specified coverage requirements. The capabilities of Eclipse Modeling Framework (EMF) to develop plugins are used by Heinecke *et al.* [11] to implement a prototype tool which is capable of importing UML ADs exported in the XMI format. The tool generates a human readable test plan for acceptance testing by combining the test cases. Although the tool is still in early stages, Heinecke *et al.* [11] claim that it shows a general feasibility for their proposed approach.

Sun *et al.* [34] used Yices SMT constraint solver to generate test cases. They proposed a process in which an AD is used to generate mutants by using some first-order mutation operators. The mutated ADs are then used to generate path constrains, which are used by the constraint solver to generate test cases. Chen *et al.* [65] proposed a directed test case generation methodology for UML ADs by translating UML AD specifications to the input language of Cadnece SMV model checker. Test cases (sequences of variable assignments) for ADs were obtained from the counter-examples produced by the model checker.

The test execution tools used in the primary studies were also mostly commercial. Boghdady *et al.* [40] suggested tools such as Test complete, IBM Rational Robot and Functional Tester, and Automated QA to execute test cases from where the results can be used to generate traceability matrices via test management tools. However, no concrete evidence on test case execution on these tools is provided. Similarly, Hartmann *et al.* [63] used a commercial tool called Compuware TestPartner to execute test cases generated via Rational Rose. Li. *et al.* [55] used a free tool called OS XCode Instruments and an open source tool called Android Robotium for test execution and post-test analysis, respectively. They used these tools for user-interface testing of some Java and Android applications.

Figure 8 presents the number of primary studies (as bubble size) with respect to types of tool support and availability of tool. The results show that in 19 out of 41 studies in which some kind of tool support was reported the primary focus had been on model editing. Moreover, most of the model editing tools are available as commercial products. Similarly, most of the test generation tools were developed in-house. Furthermore, for test execution, only one study reported free and open source tools. Therefore, there is a lack of open source, free, and free community edition tools for model editing, test generation, and test execution.

6. Threats to validity

The results of this SMS might have been affected by the following limitations:

Incomplete selection of publications: in Section 4, we have specified our systematic and unbiased selection process of the primary studies based on the guidelines given by Petersen et al. [12] as well as Kitchenham and Charters [18]. Nevertheless, it is possible that we have missed relevant literature since we only included those publications, which contain either of the search terms “activity diagram” or “test generat*” in the titles or the abstracts. Additionally, we did not contact any researcher or expert to ask about any unpublished results. We did not include gray literature such as working papers and technical reports because the quality of the gray literature is difficult to determine. It could imply that some publications could have been incorrectly excluded. In order to mitigate this risk, we searched the most common and popular electronic databases in which a large number of journals, conference and workshop proceedings, and book chapters in the software engineering field are indexed [26]. Further, we employed backward snowballing to include additional potential studies.

Inaccuracy of data extraction: owing to the subjectiveness of the data extraction process, we might have collected inaccurate results. In order to alleviate this risk, we have established the data extraction form in Section 4.6 to specify what data should be extracted from the primary studies against each research question. Furthermore, two researchers extracted the data from the studies while two other researchers validated the extracted data.

Predatory publishers: we did not exclude any primary studies on the suspicion of being published by a *predatory* publisher [76]. To the best of our knowledge, there is no widely accepted list of predatory publishers [77, 78]. Instead, we rigorously evaluated the quality of the primary studies according to our quality assessment criteria presented in Section 4.5 and excluded ten primary studies.

Bias in the quality assessment: evaluating the quality level of the primary studies objectively was a challenging task. We have attempted to complete this task fairly by clearly defining our quality assessment criteria in Section 4.5. In order to reduce the risk of underestimating the quality of different studies and, consequently excluding them, each study was scored independently by three researchers.

7. Conclusions

The objective of this Systematic Mapping Study (SMS) was to collect and classify the current body of evidence regarding test case generation approaches using the Unified Modeling Language (UML) Activity Diagrams (ADs). By applying the SMS process, we identified and analyzed 41 studies, and answered nine research question formulated in the protocol. Moreover, during this study, we have observed the following:

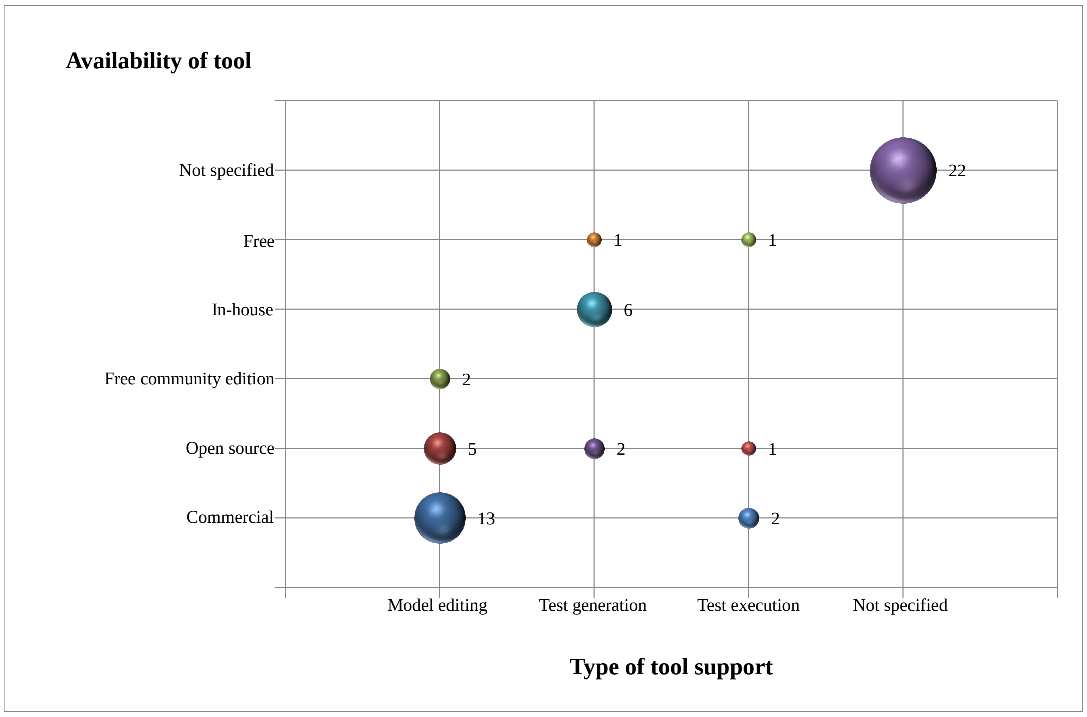

Figure 8: Number of primary studies (as bubble size) with respect to types of tool support and availability of tool

Non-functional testing: UML ADs have not been used for testing the non-functional requirements of the SUT, even though a system encounters more non-functional related problems than functional related in the real world [79].

Insufficient evaluation and validation: Most of the proposed approaches are evaluated insufficiently for a practitioner to make decisions based on the research alone. In several studies, only one characteristic of the problem is considered such as concurrency or loops in the diagram. Moreover, only in five primary studies [65, 63, 64, 51, 34], the proposed approaches are validated using industrial case studies. Furthermore, in 26 primary studies, the same academic example of an automated-teller machine (ATM) has been used for validation. For a practitioner to make use of these results, the proposed approaches should be evaluated and validated through realistic case studies.

Domain specific: In most of the proposed approaches, the targeted application domain is very restricted to be readily utilized directly by the practitioners, for example, the approach presented by Chen *et al.* [49] is only applicable to Java programs.

Holistic approach: The vast majority of the analyzed primary studies focus on specific parts of the Model-Based Testing (MBT) process without detailing all the steps of the process or how presented approaches are integrated in the development process w.r.t model provenance or requirements traceability across the testing process. In order to be applicable to industrial settings, such a holistic approach is necessary.

In addition to the aforementioned observations, we have also highlighted our findings at the end of every research question. The results of this study can be beneficial for both researchers and practitioners. For researchers, our study provides an overview of the area and highlights some gaps and directions for future research. Similarly, practitioners interested in UML AD based testing techniques and tools can use our results to explore and build suitable tool-chains according to their business needs.

8. Acknowledgments

This work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737494. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland, Czech Republic.

References

[1] P. Ammann, J. Offutt, Introduction to software testing, Cambridge University Press, 2008.

[2] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, Z. Guoliang, Generating test cases from UML activity diagram based on Gray-box method, in: Software Engineering Conf., 2004. 11th Asia-Pacific, 2004, pp. 284–291. `doi:10.1109/APSEC.2004.55`.

[3] B. Beizer, Software Testing Techniques (2Nd Ed.), Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[4] I. K. El-Far, J. A. Whittaker, Model-Based Software Testing, John Wiley & Sons, Inc., 2002. `doi:10.1002/0471028959.sof207`.
URL `http://dx.doi.org/10.1002/0471028959.sof207`

[5] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[6] C. E. Williams, Software testing and the UML, in: Proceedings of the 10th Int'l. Symposium on Software Reliability Engineering, Boca Raton, Florida, 1999, p. 2.

[7] P. C. Jorgensen, Software testing: a craftsmans approach, CRC press, 2016.

[8] H. S. Hong, Y. G. Kim, S. D. Cha, D. H. Bae, H. Ural, A test sequence selection method for statecharts, Software Testing, Verification and Reliability 10 (4) (2000) 203–227. `doi:10.1002/1099-1689(200012)10:4<203::AID-STVR212>3.0.CO;2-2`.
URL `http://dx.doi.org/10.1002/1099-1689(200012)10:4<203::AID-STVR212>3.0.CO;2-2`

[9] J. A. Whittaker, M. G. Thomason, A markov chain model for statistical software testing, IEEE Transactions on Software engineering 20 (10) (1994) 812–824.

[10] OMG, Unified Modeling Language v2.5, http://www.omg.org/spec/UML/2.5/, retrieved: August, 2016.

[11] A. Heinecke, T. Bruckmann, T. Griebe, V. Gruhn, Generating test plans for acceptance tests from UML activity diagrams, in: 2010 17th IEEE Int'l. Conf. and Workshops on Engineering of Computer Based Systems (ECBS), 2010, pp. 57–66. `doi:10.1109/ECBS.2010.14`.

[12] K. Petersen, S. Vakkalanka, L. Kuzniarz, Guidelines for conducting systematic mapping studies in software engineering: An update, Information and Software Technology 64 (2015) 1 – 18. doi:https://doi.org/10.1016/j.infsof.2015.03.007.
URL http://www.sciencedirect.com/science/article/pii/S0950584915000646

[13] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson, Systematic mapping studies in software engineering, in: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, EASE'08, BCS Learning & Development Ltd., Swindon, UK, 2008, pp. 68–77.
URL http://dl.acm.org/citation.cfm?id=2227115.2227123

[14] M. Shirole, R. Kumar, UML behavioral model based test case generation: A survey, SIGSOFT Softw. Eng. Notes 38 (4) (2013) 1–13. doi:10.1145/2492248.2492274.
URL http://doi.acm.org/10.1145/2492248.2492274

[15] R. Singh, Test case generation for object-oriented systems: A review, in: Communication Systems and Network Technologies (CSNT), 4th Int'l Conf. on, 2014, pp. 981–989. doi:10.1109/CSNT.2014.201.

[16] S. S. Priya, P. Sheba, Test case generation from UML models-a survey, in: Proc. Int'l. Conf. on Information Systems and Computing (ICISC-2013), INDIA, Vol. 3, 2013, pp. 449–459.

[17] A. Kaur, V. Vig, Systematic review of automatic test case generation by UML diagrams, Int'l. Journal of Engineering Research and Technology 1 (6 (August-2012)) (2012) 17.

[18] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, Technical Report EBSE-2007-01, Software Engineering Group, School of Computer Science and Mathematics, Keele University, Keele, Staffs, UK and Dept. of Computer Science, University of Durham, Durham, UK, (January 2007).

[19] H. Störrle, Semantics and verification of data flow in UML 2.0 activities, Electronic Notes in Theoretical Computer Science 127 (4) (2005) 35–52.

[20] OMG, UML superstructure specification v2. 0 (2005).

[21] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Softw. Test. Verif. Reliab. 22 (5) (2012) 297–312. doi:10.1002/stvr.456.
URL http://dx.doi.org/10.1002/stvr.456

[22] W. R. Adrion, M. A. Branstad, J. C. Cherniavsky, Validation, verification, and testing of computer software, ACM Comput. Surv. 14 (2) (1982) 159–192. doi:10.1145/356876.356879.
URL http://doi.acm.org/10.1145/356876.356879

[23] A. Andrews, R. France, S. Ghosh, G. Craig, Test adequacy criteria for UML design models, Software Testing, Verification and Reliability 13 (2) (2003) 95–127. doi:10.1002/stvr.270.
URL http://dx.doi.org/10.1002/stvr.270

[24] J. A. McQuillan, J. F. Power, A survey of UML-based coverage criteria for software testing, Department of Computer Science. NUI Maynooth, Co. Kildare, Ireland.

[25] S. Jalali, C. Wohlin, Systematic literature studies: database searches vs. backward snowballing, in: Proceedings of the ACM-IEEE Int'l. symposium on Empirical software engineering and measurement, ACM, 2012, pp. 29–38.

[26] T. Dybå, T. Dingsøyr, G. K. Hanssen, Applying systematic reviews to diverse study types: An experience report., in: ESEM, Vol. 7, 2007, pp. 225–234.

[27] A. Ashraf, B. Byholm, I. Porres, Distributed virtual machine consolidation: A systematic mapping study, Computer Science Review 28 (2018) 118 – 130. doi:https://doi.org/10.1016/j.cosrev.2018.02.003.
URL http://www.sciencedirect.com/science/article/pii/S1574013717300953

[28] M. Usman, E. Mendes, F. Weidt, R. Britto, Effort estimation in agile software development: A systematic literature review, in: Proceedings of the 10th International Conference on Predictive Models in Software Engineering, PROMISE '14, ACM, New York, NY, USA, 2014, pp. 82–91. doi:10.1145/2639490.2639503.
URL http://doi.acm.org/10.1145/2639490.2639503

[29] R. Chandler, C. Lam, H. Li, AD2US: an automated approach to generating usage scenarios from UML activity diagrams, in: Software Engineering Conf., 2005. APSEC '05. 12th Asia-Pacific, 2005, pp. 8 pp.–. doi:10.1109/APSEC.2005.25.

[30] D. Xu, H. Li, C. P. Lam, Using adaptive agents to automatically generate test scenarios from the uml activity diagrams, in: 12th Asia-Pacific Software Engineering Conference (APSEC'05), 2005, pp. 8 pp.–. doi:10.1109/APSEC.2005.110.

[31] S. Tiwari, A. Gupta, An Approach to Generate Safety Validation Test Cases from UML Activity Diagram, in: Software Engineering Conf. (APSEC), 20th Asia-Pacific, Vol. 1, 2013, pp. 189–198. doi:10.1109/APSEC.2013.35.

[32] X. Bai, C. P. Lam, H. Li, An approach to generate the thin-threads from the UML diagrams, in: Computer Software and Applications Conf., COMPSAC 2004. Proceedings of the 28th Annual Int'l., 2004, pp. 546–552 vol.1. doi:10.1109/CMPSAC.2004.1342893.

[33] C.-a. Sun, A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams for concurrent applications, in: Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE Int'l., 2008, pp. 160–167. doi:10.1109/COMPSAC.2008.74.

[34] H. Sun, M. Chen, M. Zhang, J. Liu, Y. Zhang, Improving defect detection ability of derived test cases based on mutated UML activity diagrams, in: IEEE 40th Annual Computer Software and Applications Conf. (COMPSAC'16), Vol. 1, 2016, pp. 275–280. doi:10.1109/COMPSAC.2016.136.

[35] P. Samuel, R. Mall, Slicing-based test case generation from UML activity diagrams, SIGSOFT Softw. Eng. Notes 34 (6) (2009) 1–14. doi:10.1145/1640162.1666579.
URL http://doi.acm.org/10.1145/1640162.1666579

[36] M. Shirole, R. Kumar, Testing for concurrency in UML diagrams, SIGSOFT Softw. Eng. Notes 37 (5) (2012) 1–8. doi:10.1145/2347696.2347712.
URL http://doi.acm.org/10.1145/2347696.2347712

[37] M. Shirole, R. Kumar, Test scenario selection for concurrency testing from UML models, in: Eighth Int'l. Conf. on Contemporary Computing (IC3), 2015, pp. 531–536. doi:10.1109/IC3.2015.7346739.

[38] A. Hettab, E. Kerkouche, A. Chaoui, A graph transformation approach for automatic test cases generation from UML activity diagrams, in: Proceedings of the Eighth Int'l. C Conf. on Computer Science and Software Engineering, ACM, 2008, pp. 88–97. doi:10.1145/2790798.2790801.
URL http://doi.acm.org/10.1145/2790798.2790801

[39] D. Kundu, D. and Samanta, A Novel Approach to Generate Test Cases from UML Activity Diagram, The Journal of Object Technology 8 (3) (2009) 65–83. doi:10.5381/jot.2009.8.3.a1.

[40] P. Boghdady, N. Badr, M. Hashim, M. Tolba, An enhanced test case generation technique based on activity diagrams, in: Int'l. Conf. on Computer Engineering Systems (ICCES), 2011, pp. 289–294. doi:10.1109/ICCES.2011.6141058.

[41] M. Khandai, A. A. Acharya, D. P. Mohapatra, Test case generation for concurrent system using UML combinational diagram, Int'l. Journal of Computer Science and Information Technologies, IJCSIT 2.

[42] V. M. Sumalatha, G. Raju, An model based test case generation technique using genetic algorithms, Int'l. Journal of Computer Science 1 (9) (2012) 46–57.

[43] P. Boghdady, N. Badr, M. Hashem, M. Tolba, Automatic generation of multi-testing types test cases using requirements-based testing, in: 7th Int'l. Conf. on Computer Engineering Systems (ICCES), 2012, pp. 249–254. doi:10.1109/ICCES.2012.6408523.

[44] M. Shirole, M. Kommuri, R. Kumar, Transition sequence exploration of UML activity diagram using evolutionary algorithm, in: Proceedings of the 5th India Software Engineering Conf., ISEC '12, ACM, 2012, pp. 97–100. doi:10.1145/2134254.2134271.
URL http://doi.acm.org/10.1145/2134254.2134271

[45] P. Mahali, A. A. Acharya, Model based test case prioritization using UML activity diagram and evolutionary algorithm, Int'l. Journal of Computer Science and Informatics 3 (2) (2013) 42–47.

[46] F. M. Nejad, R. Akbari, M. M. Dejam, Using memetic algorithms for test case prioritization in model based software testing, in: 1st Conf. on Swarm Intelligence and Evolutionary Computation (CSIEC), 2016, pp. 142–147. doi:10.1109/CSIEC.2016.7482129.

[47] L. Li, X. Li, T. He, J. Xiong, Extenics-based Test Case Generation for UML Activity Diagram, in: Y. Shi, Y. Xi, P. Wolcott, Y. Tian, J. Li, D. Berg, Z. Chen, E. HerreraViedma, G. Kou, H. Lee, Y. Peng, L. Yu (Eds.), First Int'l. Conf. on Information Technology and Quantitative Management, Vol. 17, Elsevier, 2013, pp. 1186–1193.

[48] A. Nayak, D. Samanta, Synthesis of test scenarios using UML activity diagrams, Software and Systems Modeling 10 (1) (2011) 63–89. doi:10.1007/s10270-009-0133-4.

[49] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, X. Li, Uml activity diagram-based automatic test case generation for java programs, Computer Journal 52 (5) (2009) 545–556. doi:10.1093/comjnl/bxm057.

[50] S. P. G., H. Mohanty, Automated scenario generation based on uml activity diagrams, in: 2008 International Conference on Information Technology, 2008, pp. 209–214. doi:10.1109/ICIT.2008.52.

[51] U. Farooq, C. Lam, A max-min multi objective technique to optimize model based test suite, in: 10th ACIS Int'l. Conf. on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing. SNPD '09, 2009, pp. 569–574. doi:10.1109/SNPD.2009.33.

[52] C. a Sun, B. Zhang, J. Li, TSGen: A UML Activity Diagram-Based Test Scenario Generation Tool, in: Int'l. Conf. on Computational Science and Engineering, CSE '09, Vol. 2, 2009, pp. 853–858. doi:10.1109/CSE.2009.99.

[53] P. Sapna, H. Mohanty, Using similarity measures for test scenario selection, in: Int'l. Conf. on Industrial and Information Systems (ICIIS' 09), 2009, pp. 386–391. doi:10.1109/ICIINFS.2009.5429829.

[54] N. Ye, X. Chen, W. Ding, P. Jiang, L. Bu, X. Li, Regression test cases generation based on automatic model revision, in: 2012 Sixth Int'l. Symposium on Theoretical Aspects of Software Engineering (TASE), 2012, pp. 127–134. doi:10.1109/TASE.2012.31.

[55] A. Li, Z. Qin, M. Chen, J. Liu, ADAutomation: An Activity Diagram Based Automated GUI Testing Framework for Smartphone Applications, in: Eighth Int'l. Conf. on Software Security and Reliability, IEEE, 2014, pp. 68–77. doi:10.1109/SERE.2014.20.

[56] B. Lei, L. Wang, X. Li, Uml activity diagram based testing of java concurrent programs for data race and inconsistency, in: 2008 1st International Conference on Software Testing, Verification, and Validation, 2008, pp. 200–209. doi:10.1109/ICST.2008.64.

[57] A. Gantait, Test case generation and prioritization from UML models, in: Second Int'l. Conf. on Emerging Applications of Information Technology (EAIT), 2011, pp. 345–350. doi:10.1109/EAIT.2011.63.

[58] Q. Yuan, J. Wu, C. Liu, L. Zhang, A model driven approach toward business process test case generation, in: 10th Int'l. Symposium on Web Site Evolution, 2008. WSE 2008, 2008, pp. 41–44. doi:10.1109/WSE.2008.4655394.

[59] S. Tiwari, A. Gupta, An approach of generating test requirements for Agile software development, in: Proceedings of the 8th India Software Engineering Conf., ISEC '15, ACM, 2015, pp. 186–195. doi:10.1145/2723742.2723761.
URL http://doi.acm.org/10.1145/2723742.2723761

[60] M. Akour, B. Falah, K. Kaddouri, ADBT Frame Work as a Testing Technique: An Improvement in Comparison with Traditional Model Based Testing, Int'l. Journal of Advance Computer Science and Applications 7 (5) (2016) 7–12.

[61] S. Kansomkeat, P. Thiket, J. Offutt, Generating test cases from UML activity diagrams using the condition-classification tree method, in: 2010 2nd Int'l. Conf. on Software Technology and Engineering (ICSTE), Vol. 1, 2010, pp. 62–66. doi:10.1109/ICSTE.2010.5608913.

[62] C.-a. Sun, Y. Zhao, L. Pan, X. He, D. Towey, A transformation-based approach to testing concurrent programs using UML activity diagrams, Software-Practice & Experience 46 (4) (2016) 551–576, spe.2324. doi:10.1002/spe.2324.

URL http://dx.doi.org/10.1002/spe.2324

[63] J. Hartmann, M. Vieira, H. Foster, A. Ruder, A UML-based approach to system testing, Innovations in Systems and Software Engineering 1 (1) (2005) 12–24. doi:10.1007/s11334-005-0006-0.
URL http://link.springer.com/article/10.1007/s11334-005-0006-0

[64] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, J. Kazmeier, Automation of GUI testing using a model-driven approach, in: Proceedings of the 2006 Int'l. Workshop on Automation of Software Test, AST '06, ACM, 2006, pp. 9–14. doi:10.1145/1138929.1138932.
URL http://doi.acm.org/10.1145/1138929.1138932

[65] M. Chen, P. Mishra, D. Kalita, Efficient test case generation for validation of UML activity diagrams, Design Automation for Embedded Systems 14 (2) (2010) 105–130. doi:10.1007/s10617-010-9052-4.

[66] U. Farooq, C. P. Lam, H. Li, Towards automated test sequence generation, in: 19th Australian Conf. on Software Engineering (aswec 2008), 2008, pp. 441–450. doi:10.1109/ASWEC.2008.4483233.

[67] M. Balcer, W. Hasling, T. Ostrand, Automatic generation of test scripts from formal test specifications, in: Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, TAV3, ACM, New York, NY, USA, 1989, pp. 210–218. doi:10.1145/75308.75332.
URL http://doi.acm.org/10.1145/75308.75332

[68] U. Farooq, C. P. Lam, H. Li, Transformation methodology for UML 2.0 activity diagram into colored petri nets, Advances in Computer Science and Technology (2007) 128–133.

[69] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, Nusmv: A new symbolic model verifier, in: Int'l. Conf. on computer aided verification, Springer, 1999, pp. 495–499.

[70] OpenFTA, http://www.openfta.com/, retrieved: April, 2017.

[71] J. Warmer, A. Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[72] X. Fan, J. Shu, L. Liu, Q. Liang, Test case generation from UML subactivity and activity diagram, in: Electronic Commerce and Security, ISECS '09. Second Int'l. Symposium on, Vol. 2, 2009, pp. 244–248. doi:10.1109/ISECS.2009.160.

[73] Cadence, SMV Model Checker, http://www.kenmcmil.com/smv.html, retrieved: April, 2017.

[74] The Yices SMT Solver, http://yices.csl.sri.com/, retrieved: April, 2017.

[75] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879. doi:10.1109/32.57624.

[76] J. Beall, Medical publishing triage - chronicling predatory open access publishers, Annals of Medicine and Surgery 2 (2) (2013) 47 – 49. doi:https://doi.org/10.1016/S2049-0801(13)70035-9.
URL http://www.sciencedirect.com/science/article/pii/S2049080113700359

[77] K. D. Cobey, M. M. Lalu, B. Skidmore, N. Ahmadzai, A. Grudniewicz, D. Moher, What is a predatory journal?: A scoping review, F1000Research 7. doi:10.12688/f1000research.15256.2.
URL https://www.ncbi.nlm.nih.gov/pubmed/30135732

[78] M. Berger, Everything you ever wanted to know about predatory publishing but were afraid to ask, in: Proceedings of the ACRL 2017, ACRL '17, 2017.
URL http://www.ala.org/acrl/sites/ala.org.acrl/files/content/conferences/confsandpreconfs/2017/EverythingYouEverWantedtoKnowAboutPredatoryPublishing.pdf

[79] L. Chung, J. C. S. do Prado Leite, On non-functional requirements in software engineering, in: A. T. Borgida, V. K. Chaudhri, P. Giorgini, E. S. Yu (Eds.), Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 363–379. doi:10.1007/978-3-642-02463-4_19.
URL https://doi.org/10.1007/978-3-642-02463-4_19