Model-Based Traceability for System Stabilization

Christian M. Adriano, Xinlu Tong
University of California, Irvine
Department of Informatics
Irvine, CA 92697-3440, U.S.A.

{adrianoc,xinlut}@uci.edu

Abstract — Traceability is commonly adopted as an aid to manage test cases in face of changing requirements. Our approach to traceability is to help decide which tests and bugs should be prioritized in order to minimize the time necessary to execute acceptance test activities of testing, debugging and fixing. Our approach is to apply a set of models based on traceability among requirements, tests, and software components. In this paper we demonstrate the facts motivating the model and how the model is to be operated. We also offer as future work some research questions and possible extensions.

Index Terms — Model-Driven Engineering, Traceability, Bug Triage.

I. INTRODUCTION

Traceability is commonly adopted as an aid to manage test cases in face of changing requirements. Since a trace from requirements to test cases is kept, any change in the former set could be traced to the specific test cases. Hence, knowing which test cases became invalid and should be updated is a straightforward decision. Our use of traceability is quite different. We aim at a traceability model that provides data for deciding which tests and bugs should be approached first in what sequence. Our hypothesis is that model-based requirements engineering methods could provide an operational framework to help teams drive system stabilization activities (test, debugging and fixing).

We know a lot about bug localization [1], communication through bug reporting [2], bug triage [3][4], and bug delegation [5][6]. Meanwhile, we still know little about how teams optimally work on sequencing tests and bugs [7]. Nowadays decisions are mostly tacit and ad hoc. For instance, who better knows the code or feature takes the burden of deciding [5][6]. Relying on code ownership may not be possible for situations in which teams have to deal with legacy systems. This is also the case of an organizational culture that presupposes a distributed ownership over the source code (Agile teams). Hence, in the absence of ownership information, the team has to rely on qualitative data related to the tasks of the testing and bug fixing. More precisely, the features/requirements and code impacted, as well as their importance to goals defined by user or by the team. The performance of a team to efficiently stabilize a system is thus directly related to the objectivity (accuracy and precision) and relevance of the information of impact and importance.

In the next sections we describe and define the practical problem face by software teams in stabilization activities. The problem is supported by a case studied for which we conducted interviews and a survey. Then we describe the solution based on a model, data inputs and the computation of the data to suggest optimal sequences to help teams perform testing and bug fixing. We conclude the paper with an analysis and future work.

II. THE PROBLEM OF SYSTEM STABILIZATION

A. Problem Framing

Our problem framing is to understand the cycle of testing-debugging-fixing as a collaborative stabilization activity. By system stabilization we comprise the activities carried out during testing and code fixing. The goal to stabilize consists of having the application with acceptable quality to be deployed in a user acceptance environment. Acceptance quality is user agreed number and impact of defects. It does not normally mean zero defects. In other words, an application with minor not harmful defects (e.g. visual misalignment of widgets) may be considered as having acceptable quality. Figure-1 illustrates the stabilization process with fundamental activities of testing, deciding which bugs should be fixed by whom and when (Bug Triaging), debugging for fixing and finally verifying whether the fixes actually hold (retesting).

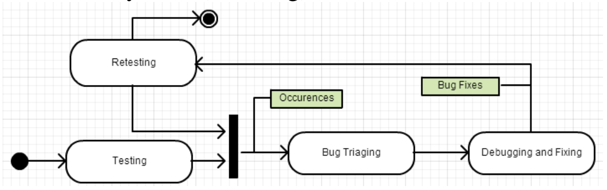


Figure-1. Typical Cycle of System Stabilization during Testing

The problem of stabilization is especially challenging for tailor made software, for which it is very difficult to predict how many and where the bugs reside after all the coding was done. Since effort and time is limited, it is paramount to wisely decide where to invest effort in bug finding. Besides that, system stabilization could be at risk when team starts modifying the code inappropriately. For instance, the failure of understanding the origin of a bug might imply on modifications resulting from unapproved change requests, invalid defects or ambiguous specifications. Moreover, the failure to recognize a proper sequence of fixing can be particularly problematic in layered or highly componentized architectures. Changing the code without being aware of code dependencies might implicate in system brittleness and code smells. Therefore, system stabilization is a real issue faced by teams in the heat of the software testing phases and sprints (e.g. agile settings). Figure-2 depicts the cycles of opening and fixing bugs during a system stabilization effort. The blue line curve represents the output of testing activities and the red line curve the output of the programmers fixing bugs. This chart demonstrates how these two activities and respective teams work synchronized and in a tightly coupled collaboration.

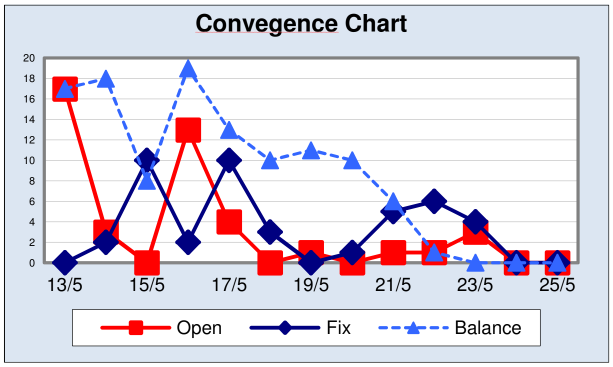


Figure-2. Example of the stabilization output for a testing phase. Red = Bugs Opened; Blue=Bugs Fixed; Dotted line = Balance (Opened-Fixed)

B. *Results from the Case Study*

The case study involved interviews and a survey. In the interviews we gathered information to refine the questions to the survey. The interviews were conducted with four senior testers and the surveyed involved twenty programmers and testers. Concerning the issue of prioritizing where to invest test effort, a tester from a military project declared that there is always an intuition where the major bugs should be at each new release or deployment. When inquired whether there were some effort to translate such intuition to a method, the tester answered negatively. This suggests that sequencing of tests as prioritization of effort is an actual though tacit practice.

A team leader of an agile team declared that time constraint is the most impacting factor for performing tests during sprints. Deciding what to test and what to fix is really challenging. A partially solution was to invest more heavily in automated testing to speed the process. The problem is that it lefts unattended the explorative test approach, which is the most effective to find bugs in new or modified areas of the system. This vision is corroborated by an anecdotal declaration of a quality manager of a major agile software environment vendor:

"We can stop a test session, create a bunch more, and head of in a new direction, which I would say more fruitful to find bugs in another area [of the software]" – QA Manager at Software Product Summit, 2011.

The interviews also demonstrated that team from different cultures have distinct approaches to sequencing and to traceability as well. The first interviewee from the military sector works under a waterfall development process and the team has 10 testers for 7 developers. The amount of testing is justified by the heterogeneity of the deployment environments. For this team, isolating testers and developers is the norm to mitigate bias (testing the features that already work or programming only what will be tested). Regarding traceability, the main concern is with user change requests and how they affect test cases and test data. Traceability is not used to decide where and how much tests to be made. The interviewee from the agile team follows a culture of distributed code ownership and relies heavily on code revision sessions. Every time a change was made on a critical code, the programmer was instructed to submit the code to a committee of experienced programmers to review it. Traceability for this team is accomplished by a tool that enforces the association of every source code commit to an issue opened in their tracking system. Issues can be new functionalities and bugs. This traceability data was used to improve the localization of a bug or a change to be implemented in the system. The traceability is neither used to decide what to test, nor the sequence of fixing.

Analyzing these two approaches to test, bug fixing and traceability, we understand they are complementary in terms of coverage, since the first gets requirements to test cases, while the second the issues to code. Nevertheless complementary, they aim at completely different needs and do not address the problem raised by our research. The first models traceability as data retrieval (concerned with finding things). The second approach models traceability as an information space (concerned with understanding things). We need a different approach, a search space concerned with finding local or global optimum to traverse the software and all its artifacts.

The survey concluded with 20 professionals confirmed some of these problems. People prioritize tests and bugs based on user satisfaction, as is demonstrated in figure-3. Therefore, the traceability from goals to bugs is strong input for sequencing tests and bugs

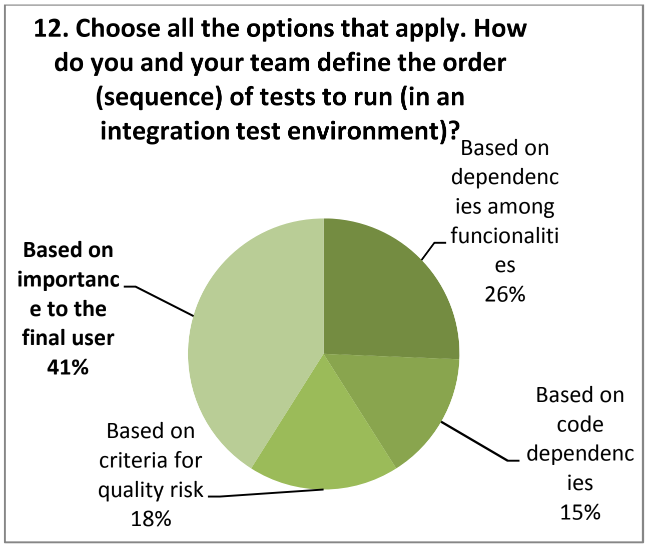


Figure-3 Test Sequencing Options Reported

Concerning the order of bug fixing, the answers also concentrated on the importance to the final user (figure-4). Such result emphasizes the use of traceability to retrieve a user as a client prioritization criterion. This type of prioritization does not take in consideration the net of artifact and code produced. We believe this concentration stem from the test prioritization decision making and not from a conscious choice made by the team.

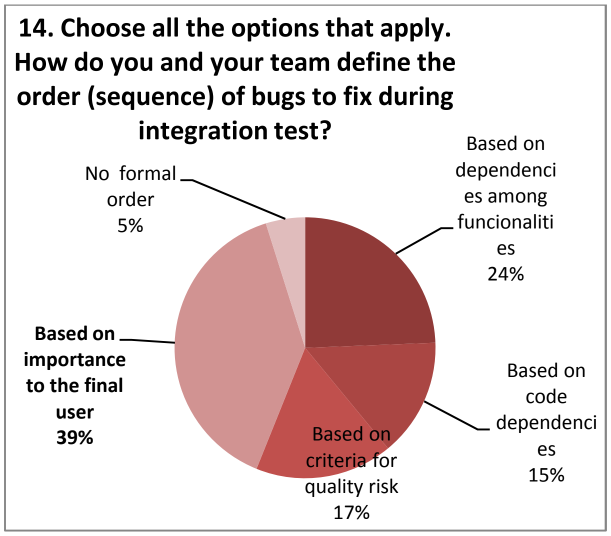


Figure-4 Bug Fix Sequencing Options Reported

The survey also demonstrated that the worst defects were the intermittent (29%, see figure-5), not the ones related to NRF (figure-4), third party code, or had a large impact on the system. Reproducing a defect implies capturing the same context of the problematic situation. Context is covered by same states and inputs used. Understanding what happens with the application in specific situations involve combining knowledge from different sources. Hence traceability, due to its connectivity nature, may be an important missing link.

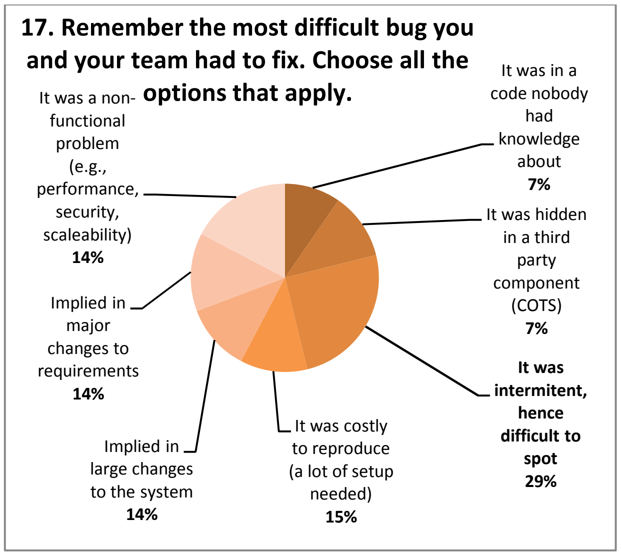


Figure-5 Root-cause of Bugs Hard to Fix

Tracking bugs involves traversing the system composed not only by code, but by all the documents produced in the requirements and design efforts. Hence, if we were to pursue the hypotheses that optimal test and fix sequences exist and are effective to system stabilization, the answer must be raised from the requirements engineering methods that create integrated maps of the knowledge about the software being grown.

III. TRACEABILITY AS A SEARCH SPACE

The decision space of what to test and fix an in which sequence is very complex. Many criteria exist to select tests and bugs, such as user satisfaction, code complexity measures, impact analysis, etc. At same time, conditions for decision change as new knowledge emerge during meetings. For example, information about cost and solution are imprecise during bug triage meetings. Therefore, previous good decision cannot be reused fully and the wrong ones only bring accidental learning to the investigation space. The software with all its artifacts is a space being grown as people acquire more knowledge [11]. It hence implies on recognizing that it is difficult to define a set of static criteria to find the optimum.

Traceability is an approach to build a map incrementally. Artifacts and its constituent parts form a map. A map can be used to localize oneself or to extract measurements (area, distances, flow, densities, etc.). We can describe a region in two metaphors – map or trajectory [12]. The map provides a global totalizing fish-eye vision of the territory based on a static perspective. Therefore, a map describes a place. Meanwhile, the trajectory provides a local subjective perspective based on movements. Trajectory describes a space. The map metaphor is convenient to understand where things are and relate to each other (entities and dependencies). Trajectory is convenient to understand how things behave in a context (execution, states, and events). For the purpose of traceability for aiding system comprehension, map metaphor is the adequate one. For the purpose of debugging an application, the trajectory metaphor is more representative of the exploration actions performed during testing and debugging. Delimit regions by the possibility of traversing or impediments of so. The former represents a space like metaphor, the latter a place like metaphor. Examples of the former are method calls and shared objects. Examples of the second are variable scopes and packages.

IV. SURVEY OF TRACEABILITY RESEARCH

A. *What Is Traceability*

In requirements engineering, traceability is an effective bridge that aligns system evolution with changing stakeholder needs. It also helps uncover unexpected problems, provide innovative opportunities, and lays the groundwork for corporate knowledge management [13]. Also, traceability aids project managers in verification, cost reduction, accountability and change management [14]. According to Gotel and Finkelstein's paper [15], the definition of traceable software is:

"Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)." [15]

From this definition, we know that traceability is an important factor throughout the whole process of software development, and the way we use to achieve traceability is to create links that connect different software artifacts.

Traceability links indicate various relationships between and within certain artifacts. There are several ways to categorize such relationships. According to the roadmap paper of software traceability [16], traceability relations denote overlap, satisfiability, dependency, evolution, generalization and refinement relations. It also represents the conflict and rationalization associations between software artifacts or contribution relations between artifacts and the stakeholders [16]. Among them, dependency and contribution relations can be directly used for system stabilization. Dependency relations reflect the dependencies and concurrency between modules and the robustness of the system. Contribution relations reflect the ownership of code and the relationships between stakeholders and artifacts, which show the relevance of a certain part of the system with user satisfaction.

Traceability seems to be merely criteria for system development just like reliabilities, or learnability; but actually, with this capability of tracing, many problems can be solved. According to Wieringa's introduction to traceability [17], the need for traceability falls into four aspects. For project management, traceability can help to estimate the impact of a change in requirements, to discover conflicts between requirements earlier and to bring reduced development time and effort for future systems because of the reuse of past implementation decisions. For customers, traceability can help to evaluate the quality of the product with respect to the user requirements and acceptance testing can refer directly to the user requirements being tested for. For designers, traceability can help to more easily verify the satisfaction from design to requirements. For maintainers, they can estimate the impact of a change in requirements on the implementation [17].

For reference, there are many insightful and thorough surveys introducing and evaluating different traceability techniques. The seminal paper of Gotel and Finkelstein [15] investigated and discussed the underlying nature of the requirements traceability problem. Other papers ([18][19]) discussed traceability techniques for model-driven engineering. Bashir and Qadir's survey [20] compared existing traceability techniques and revealed problems in them.

B. *Traceability Techniques*

To achieve traceability, There are many existing techniques that requirements engineers can choose from, including requirements matrices [21]; keyphrase dependencies [22]; reference model [23]; hypertext [24]; integration documents [25]; constraint networks [26]; goal-centric traceability [27]; value-based traceability [28] [29]. In addition, Tsumaki and Morisawa proposed a framework of traceability using UML [30], a method that supports links between UML models. It is improved by [31] and later by [32].

However, such tracing techniques mostly incur in overhead due to manually creating and maintaining traceability links; thus they are not widespread [33]; and only large companies mandated by software process standards such as CMMI or ISO 15504 end up adopting traceability techniques [34]. Also, different stakeholders in the software development process have different traceability goals [18]. Moreover, Ramesh and Jarke [23] suggested that there are two types of groups focusing on different levels of traceability information, one of which is low-end and another high-end [23]. In order to use traceability information to improve quality and efficiency of testing and debugging, we need more detailed traceability information for high-end users to ensure the precision of tracing.

Recent years have seen many automation supports for traceability. There are two trends of automated tracing: information-retrieval-based traceability recovery and traceability for model-driven engineering. Based on the feasibility study of automated requirements analysis using information retrieval methods by Dag et al. [35], Antoniol et al. proposed an approach for recovering traceability links [36]. Further research efforts ([37], [38], [39] and [40]) continued this direction of investigation. Likewise, Kagdi et al. [41] used text mining to explore software repositories and link co-changing artifacts.

The emergence of model-driven engineering technologies was to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively [42]. With a transition layer of abstraction, software development becomes closer to design intent and requirements. Also, Model-driven development provides an opportunity to automate both the creation and discovery of traceability relationships, and to maintain consistency among the heterogeneous models used throughout the system-development life cycle [18]. So this can make it easy to unify the standards of traceability and its relations, but we should consider the high cost to use such a formal method.

C. *Goal-based Requirements Engineering*

A goal is an objective the system under consideration should achieve [43]. Using goal to elicit requirements can be intuitive since it captures the rationale of requirements analysis. A goal refinement tree provides traceability links from high-level strategic objectives to low-level technical requirements [43]. In this way, we can know from traceability information that a project deviates from the ultimate objective or not. Besides that, we can find the requirements that customers care most and make decisions for bug prioritization. To support goal-based requirements engineering, there are various modeling techniques that we can choose, both formal and informal. Among them, KAOS model [44] focuses on the refinement of goals and numerous relationships between them, and *i** model [45] targets on the intentions of actors and their interactions.

D. *Challenges and Prospect of Traceability*

Despite the time and efforts made to the development of traceability, there are still many problems and challenges in it. Traceability can be difficult to accomplish in practice, primarily because creating and maintaining traceability links is time-consuming, costly, arduous, and error prone [46][23]. Also, there exists no standards and unified framework for traceability. Different categorizations of traceability relations that have been proposed in the literature and the lack of a commonly agreed standard semantics for all these types do not provide confidence in the use of traceability techniques and do not facilitate the establishment of a common framework to allow the development of tools and techniques to support automatic (or semi-automatic) generation of these relations [22].

Under this context, researchers of traceability established the International Center of Excellence for Software Traceability (CoEST) work together on the GCT (Grand Challenge of Traceability) project [47]. It is designed to challenge and inspire people to work towards achieving a difficult, yet a significant goal [48]. It identified several sub-challenges supposed to be overcome by 2035. Moreover, TraceLab, a research environment designed to facilitate innovation and creativity, will empower future traceability research [49]. Therefore, we believe that support for traceability will be more reliable and purposed in the near future, and based on that, our proposed work will be promising and trustable.

E. *Innovation of Our Work in Face of the Literature*

Our work aims at a new use of traceability, which is using traceability model to directly provide data for deciding the sequences of test cases and bugs. To the best of our knowledge, this has not been covered by any paper yet. Traceability is commonly adopted as an aid to manage test cases with respect to changing requirements and acceptance testing, so the team can directly refer to requirements to review the needs of a user, without having the cost of searching artifacts manually. Testers and developers need to review the requirements from time to time. Furthermore, for large project it is not possible to test every component and fix all the bugs before delivery or milestone. There has to be some tradeoffs for testing and debugging. To the best of our knowledge, no prior research has been made to explore the role of traceability in such situations.

On the other hand, our proposed method will use formalized traceability models with weighted nodes that indicate the relevance and significance of traceability relations. In this way, we directly use traceability information as a search space. Automated queries and decision making will be made based on some mathematical operations on the traceability models. This is more like data mining on organized and extracted information.

As future work, we will integrate the method to an existing tool. This approach can serve as the bridge or connector between bug triaging tools, bug tracking tools, repositories, modeling tools and traceability tools.

V. Requirements for the Model

A. *Goals and Requirements*

We define here high level requirements for the model and the solutions as well.

Goal-1: The model for traceability is first built before the requirements elicitation activities and is refined through requirements, design and coding activities. Therefore, it is necessary to capture in the model the goal hierarchy provided in the KAOS diagram, which already provide the traceability from goals to requirements and even to some important entities.
Requirement-1: Importing the data from this model should be considered an option; hence the model should be able to model the data from KAOS.

Goal-2: The model should be flexible to accommodate different sources of traceability data, levels of granularity for traceability data, and the customization of new queries to compute sequences (based on team defined criteria).
Requirement-2: The model should be designed as a framework with extension points (hotspots) and a core to represent the dependencies.

Goal-3: The model should enable the precise and accurate attachment of the artifacts produced in later phases (e.g. diagrams and code)
Requirement-3: A formal representation of each relationship will provide the point of attachment on each artifact

Goal-4: The users could query the traceability model by using contents from typical test phase artifacts such as bug reports, logs and stack-traces.
Requirement-4: Relationships will be represented in a matrix with numerical values. Functions will construct and query the matrix. Those artifacts could also be represented in matrices or the data could be extracted from them to query the existing matrices.

Goal-5: Enable to customize dependencies based on different criteria.
Requirement-5: Relationships should be turned off (zero) or on (1) or even have weights.

B. *Assumptions and Out-of-Scope Requirements*

Traceability is a very diverse field as we saw in the survey section. Even after defining the requirements for our model we are still left with many possible ambiguities. Hence, we were concerned with discussing what we are not considering as requirements for our model. For each non-scope we declare an assumption, as follows:

- We do not aim at coping with changes of artifacts during system stabilization
    - Assumption - it is expected that during stabilization artifacts will be solid and there will be only minor issues regarding mostly to ambiguity, error or misunderstanding of artifact content. Such issues map to the bug origination investigation.

- We do not aim at generating traceability information automatically
    - Assumption: Traceability information will be entered as people create and update new artifacts. The automation support will consist

of querying the traceability data to calculate compute the sequences.

- We do not aim at visualization and navigation through dependencies
  - <u>Assumption</u>: people will already access the artifacts and will already know where the information they need is located. We recognize that it may not be true while dealing with legacy systems or having a new member being involved in the project. We do not aim at solving the problem of providing better access to artifacts. Requirements traceability tools are available, although not used.

After having stated all the requirements and non-requirements, we pass to the actual model definition.

VI. The Model

A. *Design Rationale (Options considered)*

The design rationale involved two main sets of questions and three options for each. The option adopted is the third option for both questions.

- How traceability data can be obtained?
  - <u>Option-1</u>: Import from existing tools and enrich data
  - <u>Option-2</u>: Create new traceability data targeted to aid stabilization
  - <u>Option-3</u>: Glean it from annotations (tags) made by the team over all artifacts. Annotations could also be extracted by processing the comments and method names in code and comparing them against other artifacts.

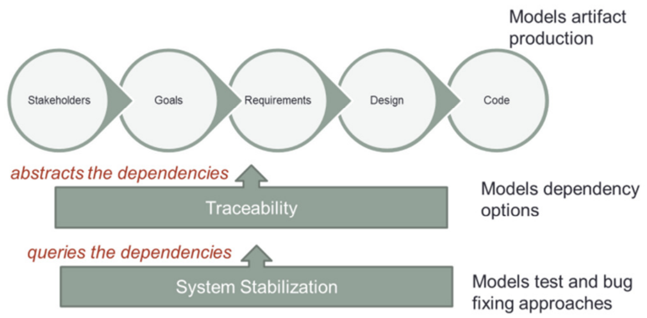
We discarded options 1 and 2 in order not to reinvent the way traceability is already created.

- How is granularity defined to effectively help in system stabilization? I.e., how to define what is traceable?
  - <u>Option-1</u>: Fixed granularity based on the traceability model defined before requirements elicitation.
  - <u>Option-2</u>: Evolving granularity based on decision points at every artifact construction
  - <u>Option-3</u>: Hierarchical granularity enables the establishment of standard categories which may be extended to accommodate smaller grains. The team can create new tags as their understanding of the software and the domain improves.

We discarded option-1 because it is the current status quo of traceability utilization, which does not solve the problem of sequencing. The option-2 was discarded because it disregards the fact that the KAOS model already provides the basic categories to trace back artifacts.

B. *Diagram Representation*

The diagram in figure-6 demonstrates how the traceability model fits within the layers of the process of producing artifacts and the process of system stabilization. The production of artifacts comprises all activities from goal modeling to code development. The system stabilization comprises planning tests, executing tests, bug triage meetings, debugging and fixing. We did not include deployment and change management activities as relevant to traceability at this point of research.



Models artifact production
Stakeholders
Goals
Requirements
Design
Code
abstracts the dependencies
Traceability
Models dependency options
queries the dependencies
System Stabilization
Models test and bug fixing approaches

Figure-6 Traceability in the context of layered processes

C. *Formal Modeling Decisions*

Models can be visual, textual, informal or formal. We chose for a textual and formal model due to the following three reasons. A formal model would enable the verification during the specification of the data mapping between existing traceability data and our solution. If new artifacts or parts of it were created, the model could expedite the analyses of impact on the algorithms consuming the traceability information. Ultimately, the formal specification could serve as an abstraction of the implementation useful to design new traceability options, such as tertiary and weighted dependencies. The formal representation provides a more intuitive base for logical reasoning on the unfolding consequences to data import, matrix representation (multi-dimensional) and querying and traversing the data structures.

As choice for data structure we decided to adopt the Design Structured Matrix (DSM) [8] method due to the expressiveness and the amount of ongoing research around it. DSM has been shown useful to specify products with complex dependencies [9] and support designers to analyze change impacts. It is also very useful to detect circular dependencies and to understand how parts are clustered, and therefore support decision making on modularizing the development process itself [10]. Such aspects are still not part of this research, but we plan to investigate their underpinnings in future work.

The matrices we adopted are bi-dimensional and combine artifacts from two families only. Therefore, the matrix is constrained by an assumption of sequential process of production of such artifacts. We acknowledge this oversimplification, which do not even represent the waterfall process. In the future we plan to experiment with multi-dimensional matrices which stem from considering dependencies as clusters. In other words, this means an artifact depending at same time on more than one artifact.

D. *Definitions*

Below we define the entities that are essential part of the traceability model.

- **Artifact (A)** = any document, diagram or code part of software project.
- **Artifact Family (A.)** = goal, requirement, design, code, test case, bug report
- **Traceable Atom (T)** = part of an artifact relevant to be related to other parts from other or same artifact. It is the smallest unit for representing artifact content, which could be lines or labels.
- **Design Structured Matrix (DSM)** = represents all dependencies among two or more different Families of Artifacts.
- **Dependency** = a relation between two Traceable
- **Sequence** = a set of Dependencies

Functions:

- **Make Dependency (MD)** = creates a Dependency
- **Compute Sequence (CS)** = obtains a Sequence given a DSM
- **Populate (PDSM)** = creates a DSM

E. *Symbols*

Artifact = **A**

Artifact Family:

- **GO** = Goal
- **RE** = Requirement document
- **DS** = Diagram
- **CD** = Code

Traceable = **T** (name) or **T** (line start, line end).

- **T.GO** ("goal1")
  - specifies that "goal1" is traceable
- **T.RE** ("req1",2,10)
  - specifies that from line 2 to line 10 of requirement "req1" is considered a traceable
- **T.DS** ("Strategy Pattern") = specifies that the Strategy Pattern diagram is a traceable
- **T.CD** ("class Validator") = specifies that the piece of code named "class Validator" is a traceable

Make Dependency = MD (T, T)

- MD (T.DS("Strategy Pattern", T.CD("class Validator"), 1)
- Creates a dependency with value 1 between a design traceable and a code traceable. If we put value zero, we would turn off the dependency

F. *Specifying a Traceability*

1) *Coarse Grain*

A = {File A, File B}
T.GO = A [1]
T.GO ("g1") = "Enable interconnectivity"
T.RE = A [2]
A.RE ("rq1") = "Manage sessions"
MD (AR("rq1"), AG("g1"),3)

2) *Fine Grain*

MD (T.AR("rq1","alternative condition"), T.AG("g1"),1)

VII. Sample Results of Applying Traceability

Formalized traceability models serve as a search space in our proposed work. After importing the dependencies to the DSM, we will traverse them to automatically generate sequences of tests and bugs to be fixed. Here follows three examples of sequence generation for different criteria. For requirement we mean functionalities visible to the final user.

A. *User Satisfaction Criterion*

Considering a goal priority of [A, B, C] and a DSM that relates these goals to a set of defective requirements (see figure-8). Therefore, looking at DSM for bugs versus requirements, we cascade the priority from goals and obtain the following bug sequence: ([Bz], [Bw,By],[Bx])

Goals/Requirements	Req1	Req2	Req3
A	0	0	1
B	0	1	1
C	1	0	0

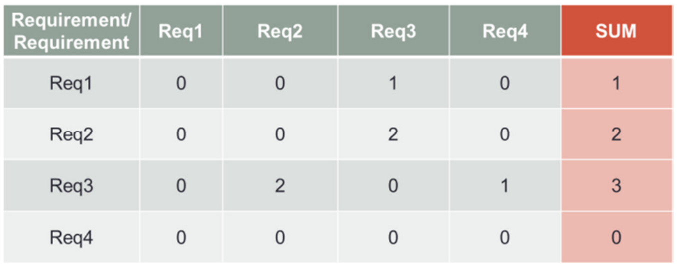
Figure-8. The dependencies between goals and requirements

Bugs/Requirements	Req1	Req2	Req3	Req4
Bx	1	0	0	0
By	0	1	0	0
Bw	0	0	1	0
Bz	0	0	0	1

Figure-9. The dependencies between bugs and requirements

B. *Robustness and Reliability Criterion*

Considering the level of dependencies on each requirement gives us the requirements more fundamental to the system, and therefore that should be fixed first. Figure-10 shows this traceability. Therefore, the resulting sequence of bugs should be as follows: ([Bw], [By],[Bx],[Bz])



Requirement/ Requirement	Req1	Req2	Req3	Req4	SUM
Req1	0	0	1	0	1
Req2	0	0	2	0	2
Req3	0	2	0	1	3
Req4	0	0	0	0	0

Figure-10. Requirement to requirement dependency

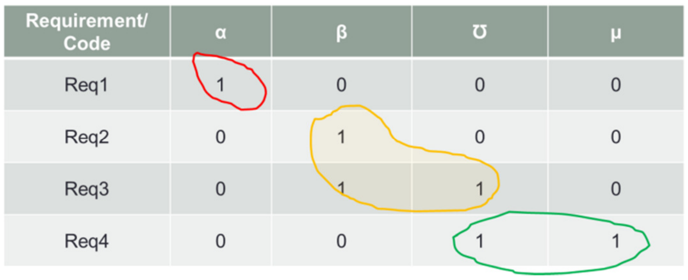
C. *Minimize Concurrency*

The objective of this criterion is to have different developers working in parallel without implying in extensive code or requirement concurrencies. It is not always possible to have zero concurrency and this criterion demonstrates that a compromise can be made by a team during bug triage.

The DSM relating code and requirements (figure-11) depicts in colored lines the sets of dependencies being accounted for the sequencing choices. In green, the sequencing suggests the bugs from requirements Req4 to be grouped. The yellow line suggests that bugs from Req2 and Req3 be grouped (so, a concurrency on the same requirement). Besides that, line yellow also implies a code concurrency on the omega component.

Such decisions need team to discuss and analyze the suggested sequences and weight the risks. For minimizing concurrency, if two bugs fall in code beta and omega, and both code beta and omega implement Req3 (as shown in Figure-11), the team may decide to put those two bugs together in a certain position of the sequence.

Therefore, querying DSMs in figure-11 and figure-9 results in the following sequence: ([Bx], [By,Bw],[Bz])



Requirement/ Code	α	β	℧	μ
Req1	1	0	0	0
Req2	0	1	0	0
Req3	0	1	1	0
Req4	0	0	1	1

Figure-11. Dependencies among bugs and code components in Greek letters

VIII. Conclusion and Future Work

The traceability model for system stabilization is a novel use for an old requirements engineering method. Our approach provides a more integrated context for its adoption. The positive outcome is to have traceability repositioned as an aid to decision making. Hence, it puts teams in a position to effectively collaborate by discussing how to maintain and consume the traceability data. Meanwhile, traceability in the current industry format is mostly aimed for individual use. The negative outcome of our research is that traceability still relies on the quality of the data produced during requirements engineering. We did not address this issue, although it is crucial to demonstrate the usefulness of the approach to system stabilization.

The future work is threefold. First, set and run experiments to investigate the effect of granularity on the precision and accuracy of the sequencing criteria. Second, extend PorchLight tool [3] to enable the use of the suggested sequences during bug triage meetings. With the tool we will be able to investigate how human factors related to motivation and intuition play a role in the decision making process. We expect that the optimality provided by the sequences would improve the quality of traceability and feedback with new criteria for sequencing. Third, we also plan to investigate more complex forms of dependencies and see how they may better represent the reality of agile or crowdsourcing based teams.

.

References

[1] Jones J. A., Harrold M. J., Stasko J., "Visualization of Test Information to Assist Fault Localization", *in proc ICSE* 2002.

[2] Breu, S., et al., Information needs in bug reports: improving cooperation between developers and users. *in proc of CSCW* 2010.

[3] Bortis G., van der Hoek A., "PorchLight: A Tag-based Approach to Bug Triaging", *in proc.* of *CSCW* 2011.

[4] Bertram D., et al., "Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams", *in proc of CSCW* 2010.

[5] Anvik J., Hiew L., Murphy, G.,C., "Who should fix this bug?", *in proc ICSE* 2006.

[6] Guo, P. J., et al., "Not My Bug! and Other Reasons for Software Bug Report Reassignment". *In proc. of CSCW* 2011.

[7] Xuan, J., et al., "Developer Prioritization in Bug Repositories", *In proc of ICSE 2012.*

[8] Browning, T.R.; , "Applying the design structure matrix to system decomposition and integration problems: a review and new directions," *Engineering Management, IEEE Transactions on* , vol.48, no.3, pp.292-306, Aug 2001

[9] Sangal N., et al., "Using dependency models to manage complex software architecture*", in proc of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, 2005

[10] Baldwin, C.Y., Clark K.B., "Design Rules – the Power of Modularity", *vol.1 ed. The MIT Press; First Edition*, Mar15, 2000

[11] Armour, P., "The Five Orders of Ignorance", *in CAMC*, vol. 43, no. 10, pp. 17-20, Oct., 2000

[12] De Certau M. and Rendall S. F, "The Practice of Everyday Life", ed. University of California Press, 1984

[13] Jarke, M. (1998). Requirements tracing. *Communications of the ACM*, *41*(12), 32-36.

[14] Watkins, R., & Neal, M. (1994). Why and how of requirements tracing. *Software, IEEE*, *11*(4), 104-106.

[15] Gotel, O. C., & Finkelstein, C. W. (1994, April). An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on* (pp. 94-101). IEEE.

[16] Spanoudakis, G., & Zisman, A. (2005). Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering*, *3*, 395-428.

[17] Wieringa, R. (1995). An introduction to requirements traceability.

[18] Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., & Shaham-Gafni, Y. (2006). Model traceability. *IBM Systems Journal*, *45*(3), 515-526.

[19] Galvao, I., & Goknil, A. (2007, October). Survey of traceability approaches in model-driven engineering. In *Enterprise Distributed*

Object Computing Conference, 2007. EDOC 2007. 11th IEEE International (pp. 313-313). IEEE.

[20] Bashir, M. F., & Qadir, M. A. (2006, December). Traceability techniques: A critical study. In *Multitopic Conference, 2006. INMIC'06. IEEE* (pp. 265-268). IEEE.

[21] Davis, A. M. (1990). Software requirements: analysis and specification. Prentice Hall Press.

[22] Jackson, J. (1991, December). A keyphrase based traceability scheme. In *Tools and Techniques for Maintaining Traceability During Design, IEE Colloquium on* (pp. 2-1). IET.

[23] Ramesh, B., & Jarke, M. (2001). Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, *27*(1), 58-93.

[24] Kaindl, H. (1993). The missing link in requirements engineering. *ACM SIGSOFT Software Engineering Notes*, *18*(2), 30-39.

[25] Lefering, M. (1993, January). An incremental integration tool between requirements engineering and programming in the large. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on* (pp. 82-89). IEEE.

[26] Bowen, J., O'Grady, P., & Smith, L. (1990). A constraint programming language for life-cycle engineering. *Artificial Intelligence in Engineering*, *5*(4), 206-220.

[27] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., & Christina, S. (2005, May). Goal-centric traceability for managing non-functional requirements. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on* (pp. 362-371). IEEE.

[28] Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., & Grünbacher, P. (Eds.). (2005). *Value-based software engineering*. Springer.

[29] Egyed, A., Biffl, S., Heindl, M., & Grünbacher, P. (2005, November). A value-based approach for understanding cost-benefit trade-offs during automated software traceability. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering* (pp. 2-7). ACM.

[30] Tsumaki, T., & Morisawa, Y. (2000). A framework of requirements tracing using UML. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific* (pp. 206-213). IEEE.

[31] Letelier, P. (2002, September). A framework for requirements traceability in UML-based projects. In *Proc. of 1st International Workshop on Traceability in Emerging Forms of Software Engineering* (pp. 173-183).

[32] Settimi, R., Cleland-Huang, J., Ben Khadra, O., Mody, J., Lukasik, W., & DePalma, C. (2004, September). Supporting software evolution through dynamically retrieving traces to UML artifacts. In *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of* (pp. 49-54). IEEE.

[33] Arkley, P., Mason, P., & Riddle, S. (2002, September). Position paper: Enabling traceability. In Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, Scotland (September 2002) (pp. 61-65).

[34] Neumuller, C., & Grunbacher, P. (2006, September). Automating software traceability in very small companies: A case study and lessons learne. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on* (pp. 145-156). IEEE.

[35] Natt och Dag, J., Regnell, B., Carlshamre, P., Andersson, M., & Karlsson, J. (2002). A feasibility study of automated natural language requirements analysis in market-driven development. *Requirements Engineering*, *7*(1), 20-33.

[36] Antoniol, G., Canfora, G., Casazza, G., & De Lucia, A. (2000). Information retrieval models for recovering traceability links between code and documentation. In *Software Maintenance, 2000. Proceedings. International Conference on* (pp. 40-49). IEEE.

[37] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, *28*(10), 970-983.

[38] Hayes, J. H., Dekhtyar, A., & Osborne, J. (2003, September). Improving requirements tracing via information retrieval. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International* (pp. 138-147). IEEE.

[39] Lucia, A. D., Fasano, F., Oliveto, R., & Tortora, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *16*(4), 13.

[40] Marcus, A., & Maletic, J. I. (2003, May). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (pp. 125-135). IEEE.

[41] Kagdi, H., Maletic, J. I., & Sharif, B. (2007, June). Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on* (pp. 145-154). IEEE.

[42] Schmidt, D. C. (2006). Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, *39*(2), 25.

[43] Van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on* (pp. 249-262). IEEE.

[44] Van Lamsweerde, A. (2000, June). Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd international conference on Software engineering* (pp. 5-19). ACM.

[45] Yu, E. S. (1997, January). Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on* (pp. 226-235). IEEE.

[46] Gotel, O., & Finkelstein, A. (1995, March). Contribution structures [Requirements artifacts]. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on* (pp. 100-107). IEEE.

[47] Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., Grünbacher, P., ... & Maletic, J. (2012). The Grand Challenge of Traceability (v1. 0). *Software and Systems Traceability*, 343-409.

[48] Cleland-Huang, J., Czauderna, A., Dekhtyar, A., Gotel, O., Hayes, J. H., Keenan, E., ... & Maeder, P. (2011, May). Grand challenges, benchmarks, and tracelab: Developing infrastructure for the software traceability research community. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering* (pp. 17-23). ACM.

[49] Keenan, E., Czauderna, A., Leach, G., Cleland-Huang, J., Shin, Y., Moritz, E., ... & Hearn, D. (2012, June). Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proceedings of the 2012 International Conference on Software Engineering* (pp. 1375-1378). IEEE Press.

BIOGRAPHIES

Christian M. Adriano (M'2005) holds a bachelor and a master in Computer Engineering from State University of Campinas. He is currently a PhD student at the University of California Irvine. Christian has maintained a Project Management Professional credential since 2008 and has more than ten years of experience in developing software for banking sector.

Xinlu Tong (M'76-SM'81-F'87) holds a bachelor in Software Engineering from Tongji University. He is currently a Master student at the University of California, Irvine.