An Evaluation of UML in Model Based Testing

Pourya Nikfard  
Advanced Informatics School (AIS)  
University Technology Malaysia (UTM)  
Kuala Lumpur, Malaysia  
npourya2@live.utm.my

Maryam Khanian Najafabadi  
Advanced Informatics School (AIS)  
University Technology Malaysia (UTM)  
Kuala Lumpur, Malaysia  
knmaryam2@live.utm.my

Babak Darvish Rouhani  
Department of Information Technology  
Payame Noor University, I.R. of IRAN  
darvishrouhani@pnu.ac.ir

Fatemeh Nikpay  
Advanced Informatics School (AIS)  
University Technology Malaysia (UTM)  
Kuala Lumpur, Malaysia  
nfatemeh6@live.utm.my

Harihodin bin Selamat  
Advanced Informatics School (AIS)  
University Technology Malaysia (UTM)  
Kuala Lumpur, Malaysia  
harihodin@ic.utm.my

Abstract- Evaluation is used in many models, across many different disciplines. It is also used for many different purposes. For models used in model-based testing, the evaluation of their testability is an important issue. The United Model Language (UML) has been applied in various areas and disciplines. Among UML, the class diagram is an important descriptive means to model requirements. But it is still lack of feasible and effective validation method which can be utilized to formalize and revise the UML class diagram. This paper investigates and presents a number of approaches. Approaches are surveyed; achievements and main issues of each approach are considered. Investigation of that classification will help researchers who are working on model testability to deliver more applicable solutions.

1. Introduction

Software testing is one of the most exclusive stages in the software development life cycle. It is costly in terms of money and time. It can show 40% of the cost of total development. Researchers and practitioners are seeking answers to reduce the cost of testing. One idea is producing tests automatically from the model or specification. Another idea is constructing systems easy to test, with the purpose of decrease this cost. It is called “design for testing” or “design for testability”. This idea depends on the observation that for a similar problem, dissimilar solutions (with dissimilar designs) can be created. Some of them are easier to test than others. As software raises more complex and starts more and more to substitute decision-makers of human in every facet of our life, software quality and reliability needs cautious concentration. Conventionally, verification and validation is the final defence against disaster caused by imperfect software development, and the software systems reliability which starts to substitute human decision makers must be high enough to avoid a disaster. But how do we conclude the critical automatic system is acceptable reliable and safe? Dynamic software testing and formal software verification are the most used two ways for
software verification. However, with complex software systems, absolute confidence on software testing often causes inappropriately high costs and other means must be found to address the effectiveness problem of software testing. In the past, much research on software testing has focused on methods for choosing effectual test data sets, variously based on specification of program, on structure of program, or on hypotheses about probable faults. Testing tries to disclose faults of software by executing the program and comparing the predictable output with the one created so that it can assure correctness.

In today’s world, the significance of delivering quality software is no longer a benefit but an essential issue. However, with the increasing complexity, software criticality and pervasiveness, main feature of assuring that it acts according to the preferred level of quality and dependability has become more critical, increasingly expensive and difficult. Furthermore, the complexity of environments and applications has considerably increased in the last couple of decades. Unfortunately, most of the software industries not only fail to deliver a quality product to their clients, but also do not understand the related attributes of quality. The software quality development still remains a guidelines matter, undocumented expert knowledge and best practices.

2. Model Testability approaches

[1] The figure summarizes a methodology that helps improve a design’s testability. The main specification for the testability analysis is the class diagram. The first step of the proposed method consists in running a testability analysis on the class diagram. This analysis detects points in the design that have to be improved for testability. These points correspond to particular configurations in the diagram that can lead to hard-to-test implementations. To run this analysis automatically on a class diagram, we need a model that can be derived from the diagram and from which it is possible to detect hard points for testability in an unambiguous way.

As a result, the testability analysis lists all the points that need to be improved in the design. It also associates a complexity measure to these points. Once the analysis has been run, it is possible to improve the design at those specific points, to reject it as too difficult to test, or to accept it as testable and implement it. The design can be improved, either by reducing coupling in the architecture, or by expressing constraints that will help the developer avoid implementing error-prone object interactions. Our suggestion is to use dedicated stereotypes on associations and dependencies specifying more clearly the type of usage that must be implemented (creation, reading.). So, when the design is implemented, the constraints are checked, and the implementation may need to be modified if the constraints are not verified [1].

[2] Focused on the evaluation of software models that are used for testing activities. They introduced the Model Quality Plan (MQP) approach for measuring quality of software models. They presented by an example how a MQP can be systematically developed for measuring the testability of UML statecharts. The MQP approach is based on a combination of the Goal Question Metric (GQM) and quality models. It consists of a top down process and a related metamodel. The process guides one how to develop a Model Quality Plan (MQP) and the metamodel states how a MQP may look like. The most important differences of their approach in respect to GQM are (1) adoption of the context characterization, (2) integration of
quality models, (3) refinement of measurement level, and (4) the formalization of their approach by an integrated UML model. Due to the documentation of a set of intermediate products and their high degree of details the initial effort for applying their approach remains heavy. Nevertheless, they are convinced that their approach is cost effective in the long term. Involving context factors tap the full potential to reuse existing quality plans. In addition, the systematic usage of context factors for the identification of information needs makes their approach more efficient. Last but not least the development effort can be considerably reduced by a dedicated tool support [2].

[3] In this paper we present a model-based testing approach where we integrate UML, UML-B and the Qtronic test generator tool, with the purpose of increasing the quality of models used for test generation via formal verification. The architectural and behavioral models of the system under test (SUT) are specified in UML and UML-B, respectively. UML-B provides UML-like visualization with precise mathematical semantics. UML-B models are developed in a stepwise manner which is then automatically translated into Event-B specifications that can be proved using theorem provers. Once the formal models are proved, they are transformed into QML which is a modeling language used by the test generation tool [3].

[4] Continuous quality assessment and improvement methodology for UML have been proposed, which is based on quality model. The quality model uses inclusion relationship for three types of model Completeness, these are: incomplete, complete and executable UML models. The quality attributes of each quality attributes are adopted from a generic quality model ISO/IEC 9126. The three types of model completeness takes into account the different level of model completeness or different level of abstraction of UML model developed during software development phases. The purpose of the quality model is to provide a way to the modeler to select appropriate methods for continuous quality assessment and improvement of UML models [4].

[5] Once the majority of the data gathering questions have been either answered out right or at least a consensus reached between primary stake holders, it is then possible to determine how testability modeling should be performed. In order to insure that the correct effort is performed during the testability analysis, it is helpful to generate a specific statement of work outlining the testability modeling effort. The following items should be included as part of the testability modeling statement of work. Once testability modeling has been performed, the data from the modeling must be reviewed, validated and implemented in order to make the most effective use of the time spent in generating the model. The following process is one method by which the model result can be generated, reviewed and incorporated into the design process [5].

[6] Extended the DbC concept to the SCT domain, and developed a new TbC technique with a key aim to bridge the gap between ordinary UML models (non-testable) and target test models (testable) and improve model-based component testability for effective UML-based SCT. Moreover, they introduced the new concept of test contract as the key testing-support mechanism, and the new concept of Contract for Testability as the principal goal of the TbC technique. They described the concept of test contracts based on basic component contracts, classified test contracts into internal and external test contracts for effective testing based on the new concept of effectual contract scope, and developed a set of TbC test criteria to realise testability improvement for achieving the CfT goals. Then, following the developed TbC working process, they put the TbC technique into practice to conduct UML-based CIT with
the CPS case study and described and discussed contract-based component test model construction, component test design, and component test generation [6].

[7] Authors in this paper first analyze and review the research and practice of existing software testability models and various factors which affect software testability in software development process and then proposes an IVF (Iteration of Vector Factorization) software testability model based on years of statistics and analysis of test data of the Software Test and Evolution Center and validates it with practice. The IVF testability model is compact and practical and can directly be applied to the practice of software engineering [7].

[8] The framework presented in this paper provides practical and operational guidelines to help assess the testability of designs modelled with the UML. These guidelines are presented in such a way that the evaluation procedure can be tailored to the specific design and test strategies employed in a specific environment. From a research viewpoint, this paper presents a number of precise hypotheses that can be investigated through empirical means. In other words, it presents a starting point theory that can be verified and refined by experimental means [8].

[9] Testability can be broken down into several individual principles. The SOCK model invented by Dave Catlett is a good way to think about this.
1. Simplicity: The simpler a component, the less expensive it is to test.
2. Observability: Exposing state (visibility and transparency)
3. Control: Can you exercise every nook and cranny of the component?
4. Knowledge of expected results: Is the observed behavior correct?
Improving each principle of SOCK improves testability. The SOCK model ensures consideration of all aspects of testability.

Testing is expensive in terms of people and equipment. We always seem to be short on test resources. While developing tests for large scale commercial products we quite often find ourselves writing a parallel product to test another. Manual testing doesn’t scale and hence we need to write a lot of automation. The following are some of the key benefits of testability:

1. Reduces the cost of testing in terms of time and resources
2. Reduces the time to diagnose unexpected behavior
3. Improves manageability and supportability of the software
4. Provides the building blocks for self-diagnosing and self-correcting software

Dromey’s generic quality model has been considered as a basis to develop the model for testability factor of aspect oriented programs. The figure shows this model which involves the following steps:
1. Identification of product properties (Aspect Oriented Software) that influences testability of software.
2. Identification of Object Oriented Design Metrics
3. A means of linking of them

On the bases of literature survey we come to know that testability is affected by encapsulation, inheritance and coupling software design properties as shown in diagram. It is the abstraction and real world modeling concepts that differentiates aspect oriented software to structural software. Three fundamental characteristics required for an aspect oriented approach: Encapsulation, Coupling and Inheritance [9].
3. Comparative evaluation
In this section we evaluate the above model testability approaches. We declare the achievement and main issue of each approach [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, and 23].

Table 1 evaluation of models

<table>
<thead>
<tr>
<th>Row</th>
<th>Authors</th>
<th>Year</th>
<th>Method</th>
<th>Achievements</th>
<th>Main Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Baudry &amp; Traon</td>
<td>2005</td>
<td>UML class diagram</td>
<td>1) Testability anti patterns 2) class interaction 3) self-usage interaction</td>
<td>1) Accountability  2) Accessibility 3) Communicativeness 4) Self-Descriptiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4) cover all object interactions 5) defines a model that can be derived from a class diagram</td>
<td>5) Availability of built-in test function 6) Retest efficiency 7) Test restartability</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6) the model is possible to detect, in an unambiguous way all the anti-patterns</td>
<td>8) not sufficient for both structural and behavioural architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7) the model computes the complexity of anti-patterns</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8) The testability measurement corresponds to the number and complexity of the anti patterns</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Voigt &amp; Engels</td>
<td>2008</td>
<td>Model Quality Plan (MQP)</td>
<td>1) measuring quality of software models 2) Goal Question Metric (GQM) 3) quality models</td>
<td>1) Accountability  2) Accessibility 3) Communicativeness 4) Not sufficient for Self-Descriptiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4) top down process adoption of the context 5) characterization integration of quality models</td>
<td>5) Availability of built-in test function 6) Retest efficiency 7) Test restartability</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6) refinement of measurement level 7) the formalization of the approach by an integrated UML model</td>
<td>8) Not sufficient for both structural and behavioural architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8) the approach is cost effective in the long term</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Malik et al.</td>
<td>2010</td>
<td>combining UML modelling with formal verification</td>
<td>1) Improve the quality of the models 2) Automated test derivation 3) Detect inconsistency in the models that were not detected by custom OCL validation rules 4) Using formal specification with UML-B and have better 5) Understanding the functionality of the SUT</td>
<td>1) Accountability  2) Accessibility 3) Communicativeness 4) Not sufficient for Self-Descriptiveness 5) Availability of built-in test function 6) Retest efficiency 7) Test restartability</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8) Not sufficient for both structural and behavioural architecture 9) Not sufficient for traceability of requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3) select appropriate methods for continuous quality assessment 4) select appropriate methods for improvement of UML models</td>
<td>8) Not sufficient for both structural and behavioural architecture</td>
</tr>
<tr>
<td></td>
<td>Author(s)</td>
<td>Year</td>
<td>Model/Technique</td>
<td>Key Points</td>
<td>Comparison Points</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------</td>
<td>------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>5</td>
<td>Emmert</td>
<td>2010</td>
<td>products test strategy</td>
<td>1) design processes of investigation, analyzing 2) alternatives, acting on the alternatives 3) to yield the optimum balance of test within a product design 4) documenting the testability 5) decisions, actions and reasons behind them 6) reduced lifecycle costs</td>
<td>1) Accountability  2) Accessibility  3) Not sufficient for Communicativeness 4) Not sufficient for Self-Descriptiveness 5) Not sufficient for Availability of built-in test function 6) Not sufficient for both structural and behavioural architecture</td>
</tr>
<tr>
<td>6</td>
<td>Zheng &amp; Bundell</td>
<td>2008</td>
<td>TbC (Test by Contract) technique</td>
<td>1) bridge the gap between ordinary UML models (non-testable) and target test models (testable) 2) improve model-based component testability for effective UML-based SCT 3) introduce the new concept of test contract 4) introduce the new concept of Contract for Testability 5) describe the concept of test contracts based on basic component contracts 6) classify test contracts into internal and external test contracts for effective testing based on the new concept of effectual contract scope 7) develop a set of TbC test criteria to realise testability improvement for achieving the CfT goals</td>
<td>1) Accountability  2) Accessibility  3) Communicativeness  4) Not sufficient for Self-Descriptiveness 5) Not sufficient for both structural and behavioural architecture</td>
</tr>
<tr>
<td>7</td>
<td>Wu et al.</td>
<td>2010</td>
<td>Software development methods</td>
<td>1) propose an IVF (Iteration of Vector Factorization) 2) software testability model validate the model with practice 3) The IVF testability model is compact and practical 4) The IVF testability model can directly be applied to the practice of software engineering</td>
<td>1) Accountability  2) Accessibility  3) Not sufficient for Communicativeness 4) Not sufficient for Self-Descriptiveness 5) Not sufficient for both structural and behavioural architecture</td>
</tr>
<tr>
<td>8</td>
<td>Mouchawrab et al.</td>
<td>2005</td>
<td>UML</td>
<td>1) presents a number of precise hypotheses that can be investigated through empirical means 2) presents a starting point theory that can be verified and refined by experimental means</td>
<td>1) Accountability  2) Accessibility  3) Not sufficient for Self-Descriptiveness 4) Not sufficient for both structural and behavioural architecture</td>
</tr>
<tr>
<td>9</td>
<td>Kumar, Sharma, &amp; Sadawarti</td>
<td>2010</td>
<td>UML</td>
<td>1) validate a model for the assessment of testability in aspect-oriented design 2) demonstrate the models' ability to estimate overall testability from design information 3) the proposed model is more practical in nature having quantitative data on testability 4) the software developer can use such data to plan and monitor testing activities 5) the tester can use testability information to determine on what module to focus during testing 6) the software developer can use testability metrics to review the design</td>
<td>1) Accountability  2) Accessibility  3) Communicativeness not sufficient for Self-Descriptiveness 4) Availability of built-in test function 5) Not sufficient for both structural and behavioural architecture</td>
</tr>
</tbody>
</table>

**4. Conclusion**

This paper has aimed to provide an overview and compare recent progress in model testability. We study and survey several approaches. But we cannot claim that these approaches are comprehensive and exhaustive. Finally, we have comparison table with different columns that compares all the approaches. The main problem with most of these approaches to testing is considering the behavioral architecture in model testability. It should be noted that although the above comparison is conducted based on some
prominent approaches, the outcome of this research is not restricted to such approaches. In other words, my considered criteria either can be served as features to be included in a newly developing system or may be applied to help generally evaluating or selecting model testability approaches. However the results of my comparison show that there is no single superior model testability approach in all cases. Therefore, deciding which approach to use in a certain scenario should be done based on its specifications and combine and present the new approach that is more comprehensive and mature is my future work.

5. Reference


[7] Ding Zhen-Guo, Research and Practice of the IVF Software Testability Testability Model, 2010 Second WRI World Congress on Software Engineering


