Model-Based Testing: The New Revolution in Software Testing

Hitesh KUMAR SHARMA, Sanjeev KUMAR SINGH, Prashant AHLAWAT
[1]University of Petroleum and Energy Studies
[2]Galgotia University Noida
[3]GITM Gurgaon
hkshitesh@gmail.com, sksingh8@gmail.com, prashantahlawat@ymail.com

The efforts spent on testing are enormous due to the continuing quest for better software quality, and the ever growing complexity of software systems. The situation is aggravated by the fact that the complexity of testing tends to grow faster than the complexity of the systems being tested, in the worst case even exponentially. Whereas development and construction methods for software allow the building of ever larger and more complex systems, there is a real danger that testing methods cannot keep pace with construction, hence these new systems cannot be sufficiently fast and thoroughly be tested. This may seriously hamper the development of future generations of software systems.
One of the new technologies to meet the challenges imposed on software testing is model-based testing. Models can be utilized in many ways throughout the product life-cycle, including: improved quality of specifications, code generation, reliability analysis, and test generation.
This paper will focus on the testing benefits from MBT methods and review some of the historical challenges that prevented model based testing and we also try to present the solutions that can overcome these challenges.
Keywords: *MBT, Test Cases, SUT, Test Suite*.

1 Introduction

"Model-based testing is a testing technique where the runtime behavior of an implementation under test is checked against predictions made by a formal specification, or model."[7]. The IEEE definition of testing is "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results." [8]. Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment. A software failure occurs when a piece of software does not perform as required and expected. In testing, the software is executed with input data, or test cases, and the output data is observed. As the complexity and size of software grows, the time and effort required to do sufficient testing grows. Manual testing is time consuming, labor-intensive and error prone. Therefore it is pressing to automate the testing effort. The testing effort can be divided into three parts: test case generation, test execution, and test evaluation.

However, the problem that has received the highest attention is test-case selection. A test case is the triplet [S, I, O] where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system. The output data produced by the execution of the software with a particular test case provides a specification of the actual program behavior. Test case generation in practice is still performed manually most of the time, since automatic test case generation approaches require formal or semi-formal specification to select test case to detect faults in the code implementation. Code based testing not an entirely satisfactory approach to generate guarantee acceptably thorough testing of modern software products. Source code is no longer the single source for selecting

test cases, and nowadays, we can apply testing techniques all along the development process, by basing test selection on different pre-code artifacts, such as requirements, specifications and design models [9],[10]. Such a model may be generated from a formal specification or may be designed by software engineers through diagrammatic tools. Code based testing has two important disadvantages. First, certain aspects of behavior of a system are difficult to extract from code but are easily obtained from design models. The state based behavior captured in a state diagram and message paths are simple examples of this. It is very difficult to extract the state model of a class from its code. On the other hand, it is usually explicitly available in the design model. Similarly, all different sequences in which messages may be interchanged among classes during the use of a software is very difficult to extract from the code, but is explicitly available in the UML sequence diagrams. Another prominent disadvantage of code based testing is very difficult to automate and code based testing overwhelmingly depends on manual test case design.

2. Process and Terminology

We use this section to fix terminology and describe the general process of model-based testing. A test suite is a finite set of test cases. A test case is a finite structure of input and expected output: a pair of input and output in the case of deterministic transformative systems, a sequence of input and output in the case of deterministic reactive systems, and a tree or a graph in the case of non-deterministic reactive systems. The input part of a test case is called test input. In general, test cases will also include additional information such as descriptions of execution conditions or applicable configurations, but we ignore these issues here.

3. Model Based Testing

A generic process of model-based testing then proceeds as follows (Fig. 1).

Step 1.

A model of the SUT is built on the grounds of requirements or existing specification documents. This model encodes the intended behavior, and it can reside at various levels of abstraction.

The most abstract variant maps each possible input to the output “no exception” or “no crash”. It can also be abstract in that it neglects certain functionality, or disregards certain quality-of-service attributes such as timing or security.

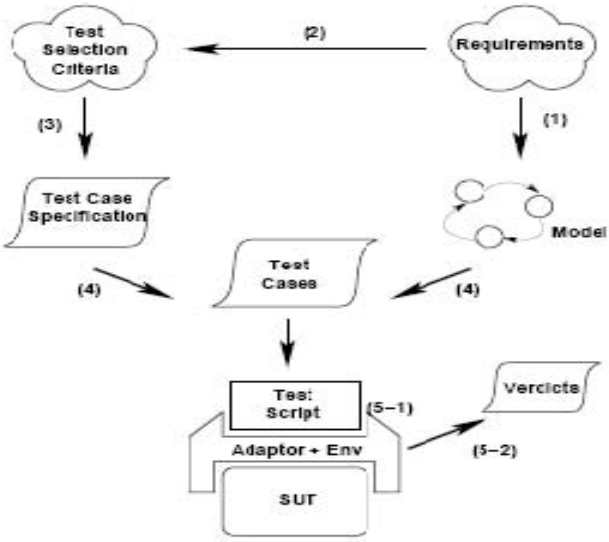

Fig. 1. The Process of Model-Based Testing

Step 2.
Test selection criteria are defined. In general, it is difficult to define a "good test case" a-priori. Arguably, a good test case is one that is likely to detect severe and likely failures at an acceptable cost, and that is helpful with identifying the underlying fault. Unfortunately, this definition is not constructive. Test selection criteria try to approximate this notion by choosing a subset of behaviors of the model. A test selection criterion possibly informally describes a test suite. In general, test selection criteria can relate to a given functionality of the system (requirements based test selection criteria), to the structure of the model (state coverage, transition coverage, def-use coverage), to stochastic characterizations such as pure randomness or user profiles, and they can also relate to a well-defined set of faults.

Step 3.
Test selection criteria are then transformed into test case specifications. Test case specifications formalize the notion of test selection criteria and render them operational: given a model and a test case specification, some automatic test case generator must be capable of deriving a test suite (see step 4). For instance, "state coverage" would translate into statements of the form "reach _" for all states _ of the (finite) state space, plus possibly further constraints on the length and number of the test cases. Each of these statements is one test case specification. The difference between a test case specification and a test suite is that the former is intensional ("fruit") while the latter is extensional ("apples, oranges, ..."): all tests are explicitly enumerated.

Step 4.
Once the model and the test case specification are defined, a test suite is generated. The set of test cases that satisfy a test case specification can be empty. Usually, however, there are many test cases that satisfy it. Test case generators then tend to pick some at random.

Step 5.
Once the test suite has been generated, the test cases are run (sometimes, in particular in the context of non-deterministic systems, generating and running tests are dove-tailed).
Running a test case includes two stages.

Step 5-1.
Recall that model and SUT reside at different levels of abstraction, and that these different levels must be bridged [2]. Executing a test case then denotes the activity of applying the concretized input part of a test case to the SUT and recording the SUT's output. Concretization of the input part of a test case is performed by a component called the adaptor. The adaptor also takes care of abstracting the output (see Fig 1).

Step 5-2.
A verdict is the result of the comparison of the output of the SUT with the expected output as provided by the test case. To this end, the output of the SUT must have been abstracted. Consider the example of testing a chip card that can compute digital signatures [7]. The verdict can take the outcomes pass, fail, and inconclusive. A test passes if expected and actual output conforms. It fails if they do not, and it is inconclusive when this decision cannot be made.

4. Importance of MBT

The first obstacle to overcome in developing tests is to determine the test target. While this may sound trivial, it is often the first place things go wrong. A description of the product or application to be tested is essential. The form the description can come in may vary from a set of call flow graphs for a voice mail system, to the user guide for a billing system's GUI. A defined set of features and / or behaviors of a product is needed in

order to define the scope of the work (both development and test). The traditional means of specifying the correct system behavior is with English prose in the form of a Requirement Specification or Functional Specification [1]. The specification, when in prose, is often incomplete - only the typical or ideal use of the feature(s) is defined, not all of the possible actions or use scenarios. This incomplete description forces the test engineer to wait until the system is delivered so that the entire context of the feature is known. When the complete context is understood, tests can be developed that will verify all of the possible remaining scenarios. Another problem with textual descriptions is that they are ambiguous, (for example "if an invalid digit is entered, it shall be handled appropriately.") The 'appropriate' action is never defined; rather, it is left to the reader's interpretation.

5. Industry importance

Modeling is a very economical means of capturing knowledge about a system and then reusing this knowledge as the system grows. For a testing team, this information is gold; what percentage of a test engineer's task is spent trying to understand what the System Under Test (SUT) should be doing? (Not just is doing.) Once this information is understood, how is it preserved for the next engineer, the next release, or change order? If you are lucky it is in the test plan, but more typically buried in a test script or just lost, waiting to be rediscovered. By constructing a model of a system that defines the systems desired behavior for specified inputs to it, a team now has a mechanism for a structured analysis of the system. Scenarios are described as a sequence of actions to the system, with the correct responses of the system also being specified. Test coverage is understood and test plans are developed in the context of the SUT, the resources available and the coverage that can be delivered. The largest benefit is in reuse; all of this work is not lost. The next test cycle can start where this one left off. If the product has new features, they can be incrementally added to the model; if the quality must be improved, the model can be improved and the tests expanded; if there are new people on the team, they can quickly come up to speed by reviewing the model.

The increased complexity of systems as well as short product release schedules makes the task of testing challenging. One of the key problems is that testing typically comes late in the project release cycle, and traditional testing is performed manually. When bugs are detected, the cost of rework and additional regression testing is costly and further impacts the product release. The increased complexity of today's software-intensive systems means that there are a potentially indefinite number of combinations of inputs and events that result in distinct system outputs and many of these combinations are often not covered by manual testing. We work with companies that have high process maturity levels, and excellent measurement data that shows that testing is more 50-75% of the total cost of a product release, yet these mature processes are not addressing this costly issue.

Test tools may not replace human intelligence in testing, but without them testing complex systems at a reasonable cost will never be possible. There are commercial products to support automated testing, most based on capture/playback mechanisms, and organizations that have tried these tools quickly realize that these approaches are still manually intensive and difficult to maintain. Even small changes to the application functionality or GUI can render a captured test session useless. But more importantly, these tools don't help test organizations figure out what tests to write, nor do they give any information about test coverage of the functionality.

6. Challenges

The real work that remains for the foreseeable future is fitting specific models (finite state machines, grammars or language-based models) to specific application domains. Often this will require new invention as mental models are transformed into actual models. Perhaps, special purpose models will be made to satisfy very specific testing requirements and more general models will be composed from any number of pre-built special-purpose models**.**

- Finding suitable abstractions is difficult
- We cannot execute partial tests

7. How can we overcome from these challenges

Fortunately, many of these problems can be resolved one way or the other with some basic skill and organization. Alternative styles of testing need to be considered where insurmountable problems that prevent productivity are encountered. We must form an understanding of how we are testing and be able to sufficiently communicate that understanding so that testing insight can be encapsulated as a model for any and all to benefit from. To achieve these goals, models must evolve from mental understanding to artifacts formatted to achieve readability and reusability. We must form an understanding of how we are testing and be able to sufficiently communicate that understanding so that testing insight can be encapsulated as a model for any and all to benefit from.

8. Conclusion

There is promising future for MBT as software becomes even more ubiquitous and quality becomes the only distinguishing factor between brands. When all vendors have the same features, the same ship schedules and the same interoperability, the only reason to buy one product over another is quality. MBT, of course, cannot and will not guarantee or even assure quality. However, its very natural, thinking through uses and test scenarios in advance while still allowing for the addition of new insights, makes it a natural choice for testers concerned about completeness, effectiveness and efficiency.

References

[1] IEEE standard for Requirements Specification (IEEE/ANSI Std. 830-1984) , IEEE Computer Society, (830-1993) IEEE Recommended Practice for Software Requirements Specifications (ANSI), IEEE Standard for Software Unit Testing (ANSI), IEEE Standard for Software Verification and Validation Plans (ANSI) found at: http://standards.ieee.org/catalog/it.html

[2] Proceedings of the Third IEEE International Symposium on Requirements Engineering, IEEE Computer Society, 1997.

[3] Paulk, M., Curtis, B., Chrissis, M.B., and Weber, C., Capability Maturity Model, Version 1.1 The Software Engineering Institute, Carnegie Mellon University. Found at: http://www.sei.cmu.edu/products/publications/96.reports/96.ar.cmm.v1.1.html

[4] ITU-T. ITU -T Recommendation Z.100: Specification and Description Language (SDL). ITU-T, Geneva, 1988. More can be found at http://www.sdl-forum.org/

[5] Spivey, M., The Z Notation: A Reference Manual, Second Edition. Prentice-Hall International, 1992.

[6] Beizer, B., *Black Box Testing*, New York, John Wiley & Sons, 1995. ISBN 0-471-12094-4.

[7] A. Pretschner, W. Prenninger, S. Wagner, C. Kuhnel, M. Baumgartner, B. Sostawa, R. Z¨olch, T. Stauner, One evaluation of model based testing and its automation, in: Proc. ICSE'05, 2005, pp. 392–401.

[8] R. Helm, I. M. Holland, and D.Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In

Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), ACM SIGPLAN Notices, 25(10):169–180, 1990.

[9] A. Pretschner, J. Philipps, Methodological Issues in Model-Based Testing, in: [29], 2005, pp. 281–291.

[10] J. Philipps, A. Pretschner, O. Slotosch, E.Aiglstorfer, S. Kriebel, K. Scholl, Model based test case generation for smart cards, in: Proc. 8th Intl. Workshop on Formal Meth. For Industrial Critical Syst., 2003, pp. 168–192.

Dr. Hitesh Kumar Sharma, The author is an Assistant Professor (Senior Scale) in University of Petroleum & Energy Studies, Dehradun. He has published 20+ research paper in International Journal and 10+ research papers in National Journals. He is Ph.D. in Computer Science & Engineering.

Dr. Sanjeev Kumar Singh: The author is an Associate Professor in Galgotias University, Noida. He has published 30+ research paper in International Journal and 15+ research papers in National Journals. He is Ph.D. in Mathematics.

Mr. Prashant Ahlawat: The author is an Assistant Professor in GITM Gurgaon. He has published 10+ research paper in International Journal and 5+ research papers in National Journals. Currently He is pursuing his Ph.D. in Computer Science & Engineering.