MODEL BASED OBJECT-ORIENTED SOFTWARE TESTING

[1]SANTOSH KUMAR SWAIN, [2]SUBHENDU KUMAR PANI, [3]DURGA PRASAD MOHAPATRA

[1]School of Computer Engineering, KIIT University, Bhubaneswar, Orissa, India-751024

[2] Department of Computer Application, RCM Autonomous, Bhubaneswar, Orissa, India -751021

[3]Department of Computer Science & Engineering, NIT, Rourkela, Orissa, India

ABSTRACT

Testing is an important phase of quality control in Software development. Software testing is necessary to produce highly reliable systems. The use of a model to describe the behavior of a system is a proven and major advantage to test. In this paper, we focus on model-based testing. The term model-based testing refers to test case derivation from a model representing software behavior. We discuss model-based approach to automatic testing of object oriented software which is carried out at the time of software development. We review the reported research result in this area and also discuss recent trends. Finally, we close with a discussion of where model-based testing fits in the present and future of software engineering.

Keywords: *Testing, Object-oriented Software, UML, Model-based testing.*

1. INTRODUCTION

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." [16]. 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 [17]. 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 [2],[3]. Such a model may be generated from a formal specification [7, 14] or may be designed by software engineers through diagrammatic tools [15]. 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.

An alternative approach is to generate test cases from requirements and specifications. These test cases are derived from analysis and design stage itself. Test case generation from design specifications has the added advantage of allowing test cases to be available early in the software development cycle, thereby making test planning more effective. Model based testing (MBT), as implied by the name itself, is the generation of test cases and evaluation of test results based on design and analysis models. This type of testing is in contrast to the traditional approach that is based solely on analysis of code and requirements specification. In traditional approaches to software testing, there are specific methodologies to select test cases based on the source code of the program to be tested. Test case design from the requirements specification is a black box approach [14], where as code-based testing is typically referred to as white box testing. Model based testing, on the other hand is referred to as the gray box testing approach.

Modern software products are often large and exhibit very complex behavior. The Object-oriented (OO) paradigm offers several benefits, such as encapsulation, abstraction, and reusability to improve the quality of software. However, at the same time, OO features also introduce new challenges for testers: interactions between objects may give rise to subtle errors that could be hard to detect. Object-oriented environment for design and implementation of software brings about new issues in software testing. This is because the above important features of an object oriented program create several testing problems and bug hazards [3]. Last decade has witnessed a very slow but steady advancement made to the testing of object-oriented systems. One of the main problems in testing object-oriented programs is test case selection. Models being simplified representations of systems are more easily amenable for use in automated test case generation. Automation of software development and testing activities on the basis of models can result in significant reductions in fault-removal, development time and the overall cost overheads.

The concept of model-based testing was originally derived from hardware testing, mainly in the telecommunications and avionics industries. Of late, the use of MBT has spread to a wide variety of software product domains. The practical applications of MBT are referred to [18]. A model is a simplified depiction of a real system. It describes a system from a certain viewpoint. Two different models of the same system may appear entirely different since they describe the system from different perspectives. For example control flow, data flow, module dependencies and program dependency graphs express very different aspects of the behavior of an implementation. A wide range of model types using a variety of specification formats, notations and languages ,such as UML, state diagrams, data flow diagrams, control flow diagrams, decision table, decision tree etc, have been established. We can roughly classify these models into formal, semiformal and informal models. Formal models have been constructed using mathematical techniques such as theory, calculus, logic, state machines, markov chains, petrinets etc. Formal models have been successfully used to automatically generate test cases. However, at present formal models are very rarely constructed in industry. Most of the models of software systems constructed in industry are semiformal in nature. A possible reason for this may be that the formal models are very hard to construct. Our focus therefore in this paper is the use of semiformal models in testing object-oriented systems.

Pretschner et al. [3] present a detailed discussion reviewing model based test generators. Barsel et al. [20] study the relationship between model and implementation coverage. The studies by Heimadahl and George[19] indicate that different test suites with the same coverage may detect fundamentally different number of errors.

This paper has been organized as follows. The next section presents an overview of various models used in object-oriented software testing. The key activities in an MBT process are discussed in section 3. Section 4 discusses the key benefits and pitfall of MBT. Section 5 focuses use of model-based testing in the present and future of software engineering. Section 6 concludes the paper.

2. MODELS USED IN SOFTWARE TESTING

In this section, we briefly review the important software models that have been used in object-oriented software testing.

2.1 UML Based Testing

Unified modeling language (UML) has over the last decade turned out to be immensely popular in both industry and academics and has been very widely used for model based testing. Since being reported in 1997, UML has undergone successive refinements. UML 2.0, the latest release of UML allows a designer to model a system using a set of nine diagrams to capture five views of the system. The use case model is the user's view of the system. A static /structural view (i.e. class diagram) is used to model the structural aspects of the system. The behavioral views depict various types of behavior of a system. For example, the state charts are used to describe the state based behavior of a system. The sequence and collaboration diagrams are used to describe the interactions that occur among various objects of a system during the operation of the system. The activity diagram represents the sequence, concurrency, and synchronization of various activities performed by the system. Behavioral models are very important in test case design, since most of the testing detect bugs that manifest during specific run of the software i.e. during a specific behavior of the software. Besides the behavioral models, it is possible to construct the implementation and environmental views of the system. The object constraint language (OCL) makes it possible to have precise models.

The work reported in [1-3, 5, 8] discuss various aspects of UML-based model testing. A vast majority of work examining MBT of object –oriented systems focuses on the use of either class or state diagrams. Both these categories of work overwhelmingly address unit testing. Class diagrams provide information about public interfaces of classes, method signatures, and the various types of relationships among classes. The state diagram-based testing focuses on making the objects all possible states and undertake all possible transitions. Several work reported recently address use of sequence diagrams, activity diagrams and collaboration diagrams in testing [9].

2.2 Finite State Machines

FSM (Finite State machines) have been used since long to capture the state –based behavior of systems. Finite state machines (also known as finite automata) have been around even before the inception of software engineering. There is a stable and mature theory of computing at the center of which are finite state machines and other variations. Using finite state models in the design and testing of computer hardware components has been long established and is considered a standard practice today. [13] was one of the earliest, generally available articles addressing the use of finite state models to design and test software components. Finite state models are an obvious fit with software testing where testers deal with the chore of constructing input sequences to supply as test data; state machines (directed graphs) are ideal models for describing sequences of inputs. This, combined with a wealth of graph traversal algorithms, makes generating tests less of a burden than manual testing. On the downside, complex software implies large state machines, which are nontrivial to construct and maintain. However, FSMs being flat representations are handicapped by the state explosion problem. State charts are an extension of FSMs that has been proposed specifically to address the shortcomings of FSMs [13].State charts are hierarchical models. Each state of a state chart may consist of lower-level state machines. Moreover they support specifications of state-level concurrency. Testing using state charts has been discussed in[21].

2.2 Markov Chains

Markov chains are stochastic models [24]. A specific class of Markov chains, the discrete-parameter, finite-state, time-homogenous, irreducible Markov chain, has been used to model the usage of software. They are structurally similar to finite state machines and can be thought of as probabilistic automata. Their primary worth has been, not only in generating tests, but also in gathering and analyzing failure data to estimate such measures as reliability and mean time to failure. The body of literature on Markov chains in testing is substantial and not always easy reading. Work on testing particular systems can be found in [22] and [23].

2.2 Grammars

Grammars have mostly been used to describe the syntax of programming and other input languages. Functionally speaking, different classes of grammars are equivalent to different forms of state machines. Sometimes, they are much easier and more compact representation for modeling certain systems such as parsers. Although they require some training, they are, thereafter, generally easy to write, review, and maintain. However, they may present some concerns when it comes to generating tests and defining coverage criteria, areas where not many articles have been published.

3. A TYPICAL MODEL-BASED TESTING PROCESS

In this section, we discuss the different activities constituting a typical MBT process.Fig.1 displays the main activities in a life cycle of a MBT process .the rectangles in Fig. 1 represent specific artifacts developed used during MBT. The ovals represent activities processes during MBT.

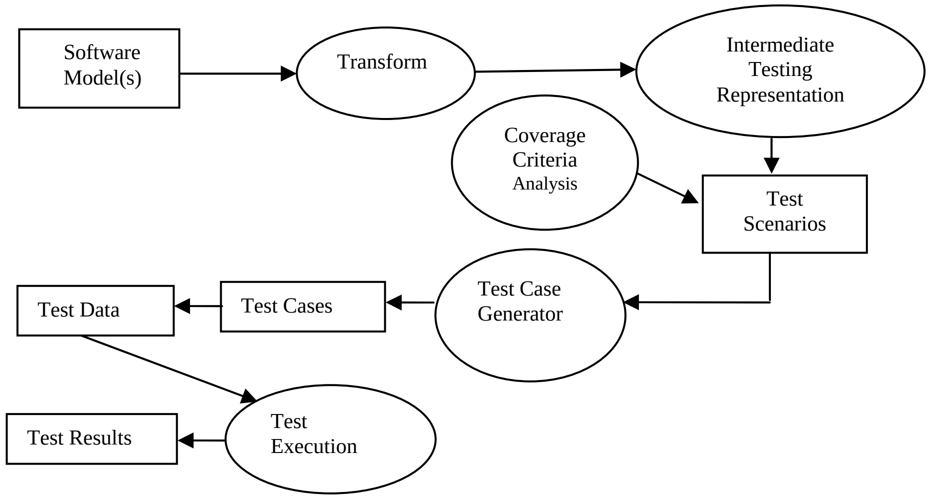


Figure 1. A Typical Model Based Testing Process

3.1 Construction of intermediate model

Several strategies have been reported to generate test cases using a variety of models. However in many cases the test cases based on more than one model type. In such cases ,it becomes necessary to first construct an integrated model based on the information present in different models.

3.2 Generation of test scenarios

The test cases generated from models are in form of sequences of test scenarios. Test scenarios specify a high level test case rather than the exact data to be input to the system. For example, in the case of FSMs, it can be the sequence in which specifies states and transitions must be undertaken to test the system-called a transition path. The sequences of different transition labels along the generated paths form the required test scenarios. Similarly from the sequence diagrams the message paths can be generated. The exact sequence messages in which the classes must interact for testing the system is shown.

3.3 Test Generation

The difficulty of generating tests from a model depends on the nature of the model. Models that are useful for testing usually possess properties that make test generation effortless and, frequently, automatable. For some models, all that is required is to go through combinations of conditions described in the model, requiring simple knowledge of combinatory. There are a variety of constraints on what constitutes a path to meet the criteria for tests. It includes having the path start and end in the starting state, restricting the number of loops or cycles in a path, and restricting the states that a path can visit.

3.4 Automatic test case execution

In certain cases the tests can even be performed manually. Manual testing is labor-intensive and time consuming. However, the generated test suite is usually too large for a manual execution. Moreover, a key point in MBT is the frequent regeneration and re-running of the test suite whenever the underlying model is changed. Accordingly achieving the full potential of MBT requires automated test execution. Usually, using the available testing interface for the software, the abstract test suite is translated into an executable test script. Automatic test case execution also involves test coverage analysis. Based on the test coverage analysis, the tests generation step may be fine tuned or different strategies may be tried out.

3.5 Test Coverage Analysis

Each test generation method targets certain specific features of the system to be tested. The extent to which the targetted features are tested can be determined using test coverage analysis[10,12]. The important coverage analysis based on a model can be the following: all model parts(or test scenarios)coverage is achieved when the test reaches every part in the model at least once. Important test coverage required based on UML models can be the following: path coverage, message path coverage, transition path coverage, scenario coverage, dataflow coverage, polymorphic coverage, inheritance coverage. Scenarios coverage is achieved when the test executes every scenario identifiable in the model at least once.

4. A CRITIQUE OF MBT

Some important MBT advantages can be summarized in the following points. It allows achieving higher test coverage. This is especially true of certain behavioral aspects which are difficult to identify in the code. Another important advantage of model–based testing is that when a code change occurs to fix a coding error, the test cases generated from the model need not change. As an example, changing the behavior of a single control in the user interface of the software makes all the test cases using that control outdated. In traditional testing scenarios, the tester has to manually search the affected test cases and update them. As even when code changes, the changed code still confirms to the model. Model based test suite generation often overcomes this problem.

However MBT does have certain restrictions and limitations. Needless to say, as with several other approaches, to reap the most benefit from MBT, substantial investment needs to be made. Skills, time, and other resources need to be allocated for making preparations, overcoming common difficulties, and working around the major drawbacks. Therefore, before embarking on a MBT endeavor, this overhead needs to be weighed against potential rewards in order to determine whether a model-based technique is sensible to the task at hand.

MBT demands certain skills of testers. They need to be familiar with the model and its underlying and supporting mathematics and theories. In the case of finite state models, this means a working knowledge of the various forms of finite state machines and a basic familiarity with formal languages, automata theory, and perhaps graph theory and elementary statistics. They need to possess expertise in tools, scripts, and programming languages necessary for various tasks. For example, in order to simulate human user input, testers need to write simulation scripts in a specialized language.

In order to save resources at various stages of the testing process, MBT requires sizeable initial effort. Selecting the type of model, partitioning system functionality into multiple parts of a model, and finally building the model are all labor-intensive tasks that can become prohibitive in magnitude without a combination of careful planning, good tools, and expert support. Finally, there are drawbacks of models that cannot be completely avoided, and workarounds need to be devised. The most prominent problem for state models (and most other similar models) is state space explosion. Briefly, models of almost any non-trivial software functionality can grow beyond management even with tool support. State explosion propagates into almost all other model-based tasks such as model maintenance, checking and review, non-random test case generation, and achieving coverage criteria. The generated test cases may in many cases get irrevalent due to the disparity between a model and its corresponding code.MBT can never displace code based testing, since models constructed during the development process lack several details of implementation that are required to generate test cases.

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.

5. MBT IN SOFTWARE ENGINEERING: TODAY AND TOMORROW

Good software testers cannot avoid models. MBT calls for explicit definition of the testing endeavor. However, software testers of today have a difficult time planning such a modeling effort. They are victims of the ad hoc model, either in advance or throughout the nature of the development process where requirements change drastically and the rule of the day is constant ship mode. Today, the scene seems to be changing. Modeling in general seems to be gaining favor; particularly in domains where quality is essential and less-than-adequate software is not an option. When modeling occurs as a part of the specification and design process, these models can be leveraged to form the basis of MBT.

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 nature, 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.

The real work that remains for the near future is fitting specific models (finite state machines, grammars or language-based models) to specific application domains. Perhaps, special purpose models will be made to satisfy very specific testing requirements and models that are more general will be composed from any number of pre-built special-purpose models. However, 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.

6. CONCLUSION

Good software testers cannot avoid models. MBT has emerged as a useful and efficient testing method for realizing adequate test coverage of systems. The usage of MBT reveals substantial benefit in terms of increase productivity and reduced development time and costs. On the other hand MBT can't replace code based testing since models are abstract higher level representations and lack of several details present in the code. It is expected that in future models shall be constructed by extracting relevant information both from the design which can automate the test case design process to a great deal.

Not surprisingly, there are no software models today that fit all intents and purposes. Consequently, for each situation decisions need to be made as to what model (or collection of models) are most suitable. There are some guidelines to be considered that are derived from earlier experiences. The choice of a model also depends on aspects of the system under test and skills of user. However, there is little or no data published that conclusively suggests that one model outstands others when more than one model is intuitively appropriate.

REFRENCES:

[1]. W. Prenninger, A. Pretschner, Abstractions for Model-Based Testing, ENTCS 116 (2005) 59–71.

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

[3]. 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.

[4]. G. Walton, J. Poore, Generating transition probabilities to support model-based software testing,Software: Practice and Experience 30 (10) (2000) 1095–1106.

[5]. A. Pretschner, O. Slotosch, E. Aiglstorfer,S. Kriebel, Model based testing for real–the inhouse card case study, J. Software Tools for Technology Transfer 5 (2-3) (2004) 140–157.

[6]. A. Pretschner, W. Prenninger, S. Wagner, C. K¨uhnel, 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.

[7]. E. Bernard, B. Legeard, X. Luck, F. Peureux, Generation of test sequences from formal specifications:GSM 11.11 standard case-study, SW Practice and Experience 34 (10) (2004) 915 – 948.

[8]. E. Farchi, A. Hartman, S. S. Pinter, Using a model-based test generator to test for standard conformance, IBM Systems Journal 41 (1) (2002) 89–110.

[9]. D. Lee, M. Yannakakis, Principles and methods of testing finite state machines — A survey, Proceedings of the IEEE 84 (2) (1996) 1090–1126.

[10]. H. Zhu, P. Hall, J. May, Software Unit Test Coverage and Adequacy, ACM Computing Surveys 29 (4) (1997) 366–427.

[11]. B. Beizer, Black-Box Testing : Techniques for Functional Testing of Software and Systems, Wiley, 1995.

[12]. C. Gaston, D. Seifert, Evaluating Coverage-Based Testing, in: [29], 2005, pp. 293–322.

[13]. A. Offutt, S. Liu, A. Abdurazik, P. Ammann, Generating test data from state-based specifications,J. Software Testing, verification and Reliability 13 (1) (2003) 25–53.

[14]. A. Pretschner, Model-Based Testing in Practice,in: Proc. Formal Methods, Vol. 3582 of SpringerLNCS, 2005, pp. 537–541.

[15]. R. V. Binder, Testing Object-Oriented Systems:Models, Patterns, and Tools, Addison-Wesley,1999.

[16]. 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-OrientedProgramming Systems, Languages, and Applications (OOPSLA ’ 90), ACM SIGPLAN Notices, 25(10):169–180, 1990.

[17]. R. Mall, Fundamentals of Software Engineering, Second ed., Prentice-Hall, Englewood Cliffs, NJ, 2003.

[18]. Ilan Gronau.Alan Hartman.Andrei Kirshin.Kenneth Nagin and Sergey Olvovsky.A methodology and architecture for automated software testing.Haifa technical report IBM Research Laborotory.MATAM ,Advanced Technology Center, Haifa 31905,Israel.2000.

[19]. M.Heimadahl and D.George, “Test suite Reduction for Model Based Tests:Effects on Test Quality and Implecations for testing” In proceedings of the 19th International Conference on Automated Software Engineering pp.176-185,2004.

[20]. A.Baresel,M.Conrad, S.Sadeghipour and j.wegener.” the interplay between model coverage and code coverage” in Eurocast, DEC2003.

[21]. D.harel”Statecharts:A visual formalism for complex systems science of computer programming,8(3):231-274,1987.

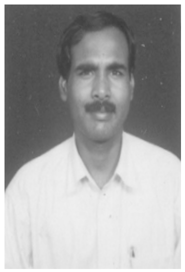
[22]. K. Agrawal and James A. Whittaker. Experiences in applying statistical testing to a real-time, embedded.software system. Proceedings of the Pacific Northwest Software Quality Conference, October 1993.

[23]. Alberto Avritzer and Brian Larson. Load testing software using deterministic state testing." Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA 1993), pp. 82-88, ACM, Cambridge, MA, USA, 1993.

[24]. J. G. Kemeny and J. L. Snell. Finite Markov chains. Springer-Verlag, New York 1976.

BIOGRAPHY: (Optional)

Santosh Kumar Swain is presently working as teaching faculty in **School of Computer Engineering, KIIT University,** KIIT, Bhubaneswar, Orissa, India**.** He has acquired his M.Tech degree from Utkal University, Bhubaneswar. He has contributed more than four papers to Journals and Proceedings. He has written one book on “Fundamentals of Computer and Programming in C”. He is a research student of KIIT University, Bhubaneswar. His interests are in Software Engineering, Object Oriented Systems, Sensor Network and Compiler Design etc.



Dr. Durga Prasad Mohapatra studied his M.Tech at **National Institute of Technology, Rourkela**, India. He has received his Ph. D from **Indian Institute of Technology, Kharagpur,India**. Currently, he is working as Associate Professor at **National Institute of Technology, Rourkela**. His special fields of interest include Software Engineering, Discrete Mathematical Structure, slicing Object-Oriented Programming. Real-time Systems and distributed computing.