International Journal of Advanced Science and Research
ISSN: 2455-4227, Impact Factor: RJIF 5.12
www.allsciencejournal.com
Volume 2; Issue 3; May 2017; Page No. 44-51

Model base testing: A review

Suruchi

Department of Computer Science and Engineering, Gian Jyoti Group of institutions, Shambhukalan, Banur-Ambala Highway, Rajpura, Punjab, India

Abstract
Model based testing has become a very popular term within the last few years. It is best known in the world of embedded systems. Software testing requires the use of a model to guide such efforts as test selection and test verification. Often, such models are implicit, existing only in the head of a human tester, applying test inputs in an ad hoc fashion. The mental model testers build encapsulates application behavior, allowing testers to understand the application's capabilities and more effectively test its range of possible behaviors. When these mental models are written down, they become sharable, reusable testing artifacts. In this case, testers are performing what has become to be known as model-based testing. Model-based testing has recently gained attention with the popularization of models (including UML) in software design and development. Finally, we close with a discussion of where model-based testing fits in the present and future of software engineering.

Keywords: software behavior models, finite state machines, state charts, unified modeling language, informal model, formal model

1. Introduction
In general the idea of MBT is to create functional test models based on requirements. The requirements are thoroughly reviewed by creating test models. Once created, these test models are used for generating test cases. Generated test cases can be used for manual and/or automated test execution.
The definition of MBT in Wikipedia[6] is the following:
"Model-based testing is software testing in which test cases are derived in whole or in part from a model that describes some (usually functional) aspects of the system under test (SUT)."

A distinction can be made between informal and formal MBT. The difference between formal and informal MBT is that formal MBT uses formal test models that comply to certain standard modelling rules while informal MBT doesn't use formal test models. Test cases can be automatically generated from formal test models and must be manually generated from informal test models. Paragraph Informal Model Based Testing and Formal Model Based Testing will explain informal and formal MBT in more detail.

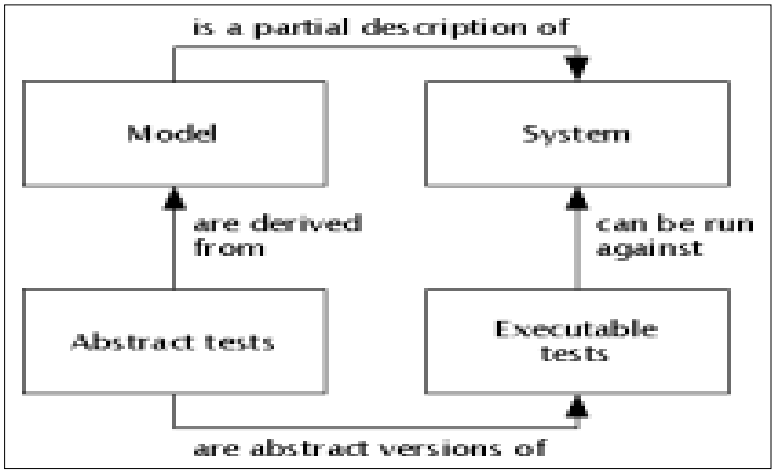

Fig 1.1: Basic Model Base testing

2. Objectives of model Base Testing
- In general the idea of MBT is to create functional test models based on requirements.
- The requirements are thoroughly reviewed by creating test models. Once created, these test models are used for generating test cases. Generated test cases can be used for manual and/or automated test execution.
- This definition indicates that the only purpose of model based testing is to derive test cases from a test model. However the creation of test models also has substantial added value as an activity of a review process.

3. Informal model based testing

Often testers start drawing some sort of test model while reading requirements. These test models help to get a better understanding of the requirements which are often only textual documents. When drawing these test models many questions will arise. Sometimes because specifications contradict each other, contain incorrect information, are unclear or, even worse, not present. By drawing test models and asking questions, the tester will gain a good insight into the system to be tested and the tester will help the functional designer to get the specifications right. Drawing such test models will help improve the quality of the requirements and will assist in finding defects at a very early phase. Creating test models is also one of the first process steps in informal model based testing, see Figure 1.

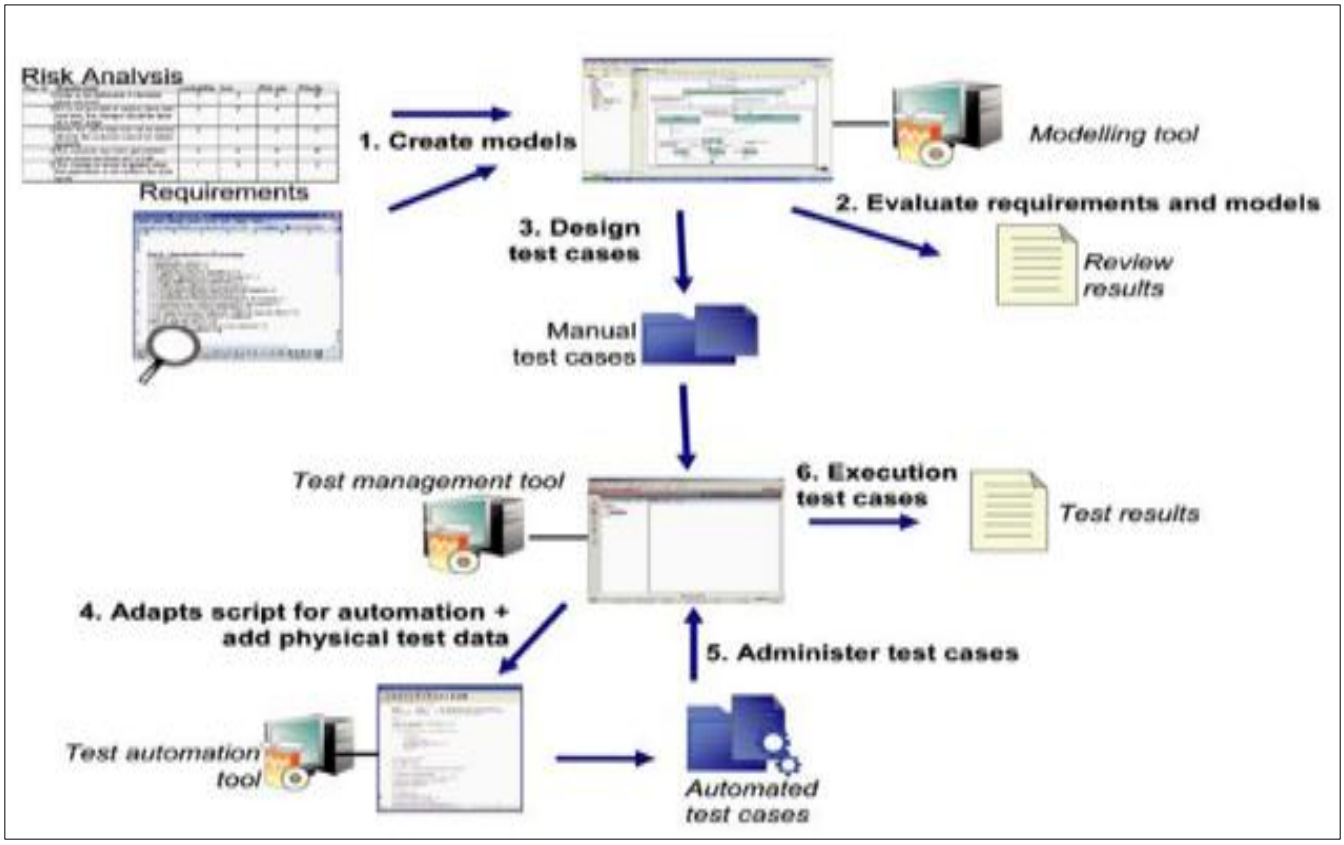

Fig 1: Process Steps informal model based testing

3.1 What exactly are the process steps of informal MBT?

Before starting to create test models it is important to consider what functionality needs to be modelled in which order. The answer to this consideration can be found in the Risk Analysis (RA) The RA should be the basis for each test project. It contains information about which specifications have a high risk score for a certain version of the system. Based on this information a choice can be made on what to model, for example only model the highest risks requirements or model the highest risks requirements first and the lower risks requirements later. The creation of a RA is preferably done in the business analysis phase but is possibly performed later.

Once the risk information is clear the modelling can start, see step 1 in Figure 1. The notation of test models used for informal model based testing doesn't have to comply with certain modelling rules. The only rule for modelling is that the test models to be created are readable for the parties/persons involved in testing. Parties or persons involved can be for example other testers, functional designers, end users etc. These parties will have to determine themselves what is a readable form. Any tool which can create test models can be used for informal model based testing. An example of an informal test model is Figure 2.

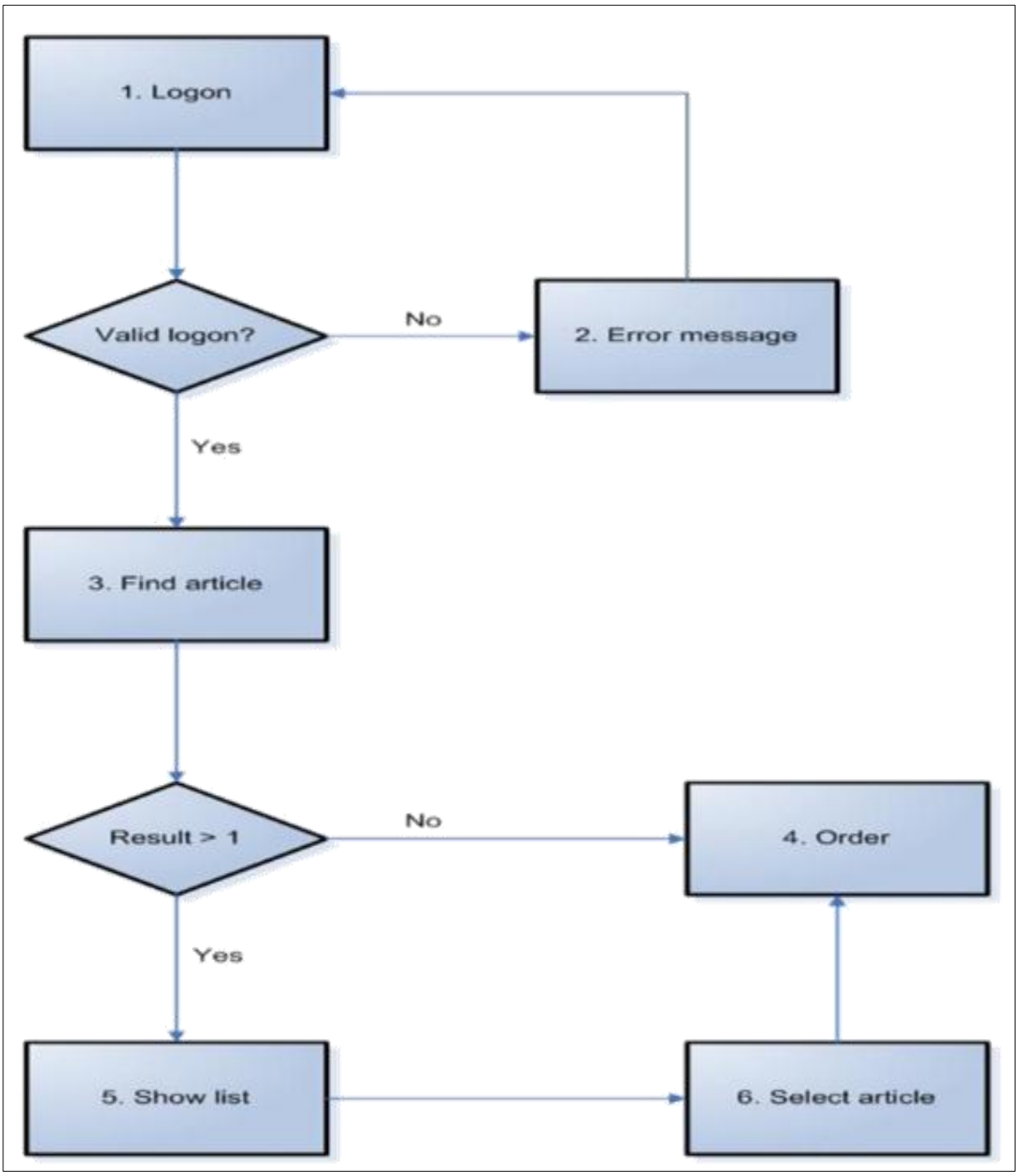

Fig 2

3.2 Formal model based testing

One of the biggest advantages of formal MBT is the possibility to automatically generate test cases based on test models. To automatically generate test cases a tool is needed which should be able to interpret the test models.

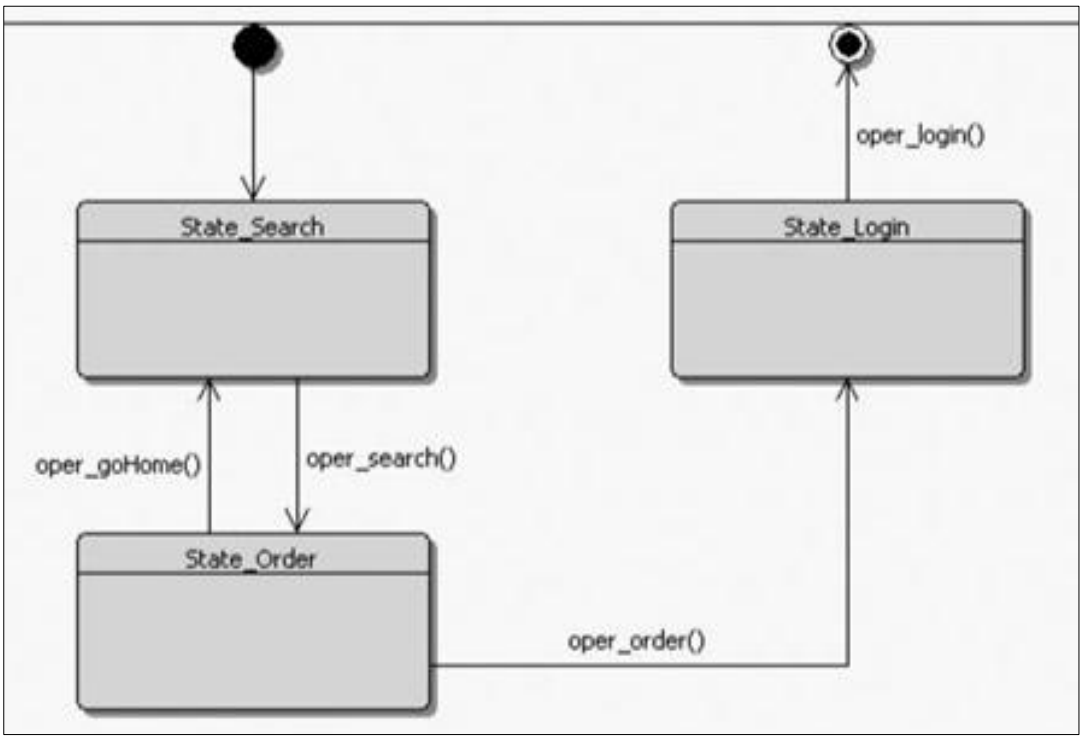

Fig 3: Formal Model

3.2.1 What exactly are the process steps of formal MBT?
As with informal MBT it's important to consider what functionality needs to be modelled and in which order. This information should be extracted from the Risk Analysis (RA). Based on this information a choice can be made for example to only model the highest risks requirements or to model the highest risks requirements first and the lower risks requirements later, see step 1 in Figure 4

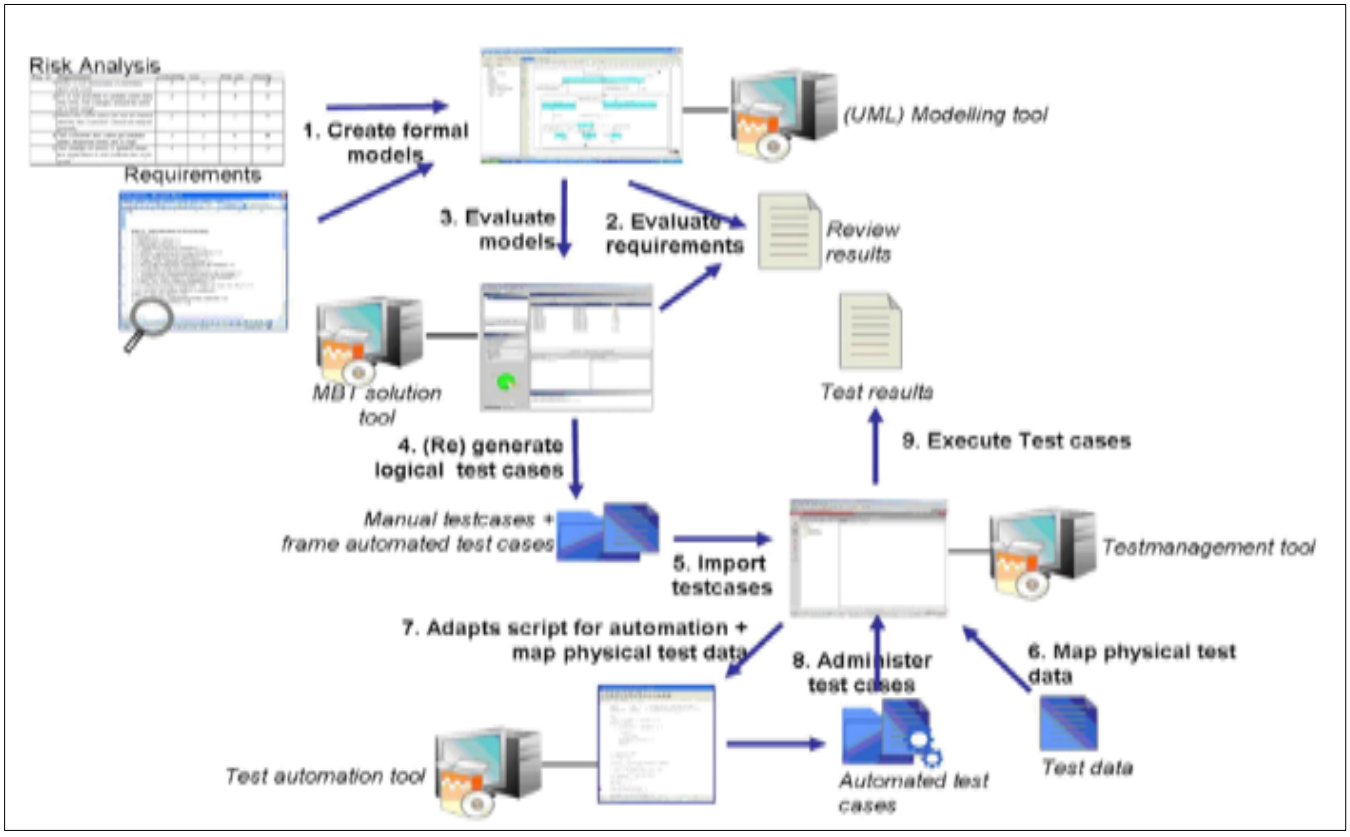

Fig 4: Process steps formal model based testing

The test models to be designed must be formal and accurate. To be able to design such test models more sophisticated modelling tools are needed, for example Borland Together or IBM Rational Software Modeler. Sometimes model based solution tools[9] also offer modelling functionality. In this white paper the assumption is made that a separate tool is used for modelling purposes.

4. Position of informal and formal MBT in the V-model

This paragraph will describe the position of informal and formal MBT in the V-model[10]. The V-model shows the relationships between each phase of the development life cycle and its associated phase of testing, see Figure 5. In the problem domain the wishes, opportunities, problems and policies are identified. These will be converted to a solution in the solution domain. The solution domain can be split up in a functional and technical domain.

Informal MBT can be used for both the functional domain and the technical domain within the solution domain. It will probably be used most in the functional domain for system, system integration and acceptance testing.

The most common kind of testing with MBT in general is functional testing but can also be some kinds of robustness testing [11]. Automated test cases generated from formal models can possibly also be used for performance testing. MBT is less useful for usability testing.

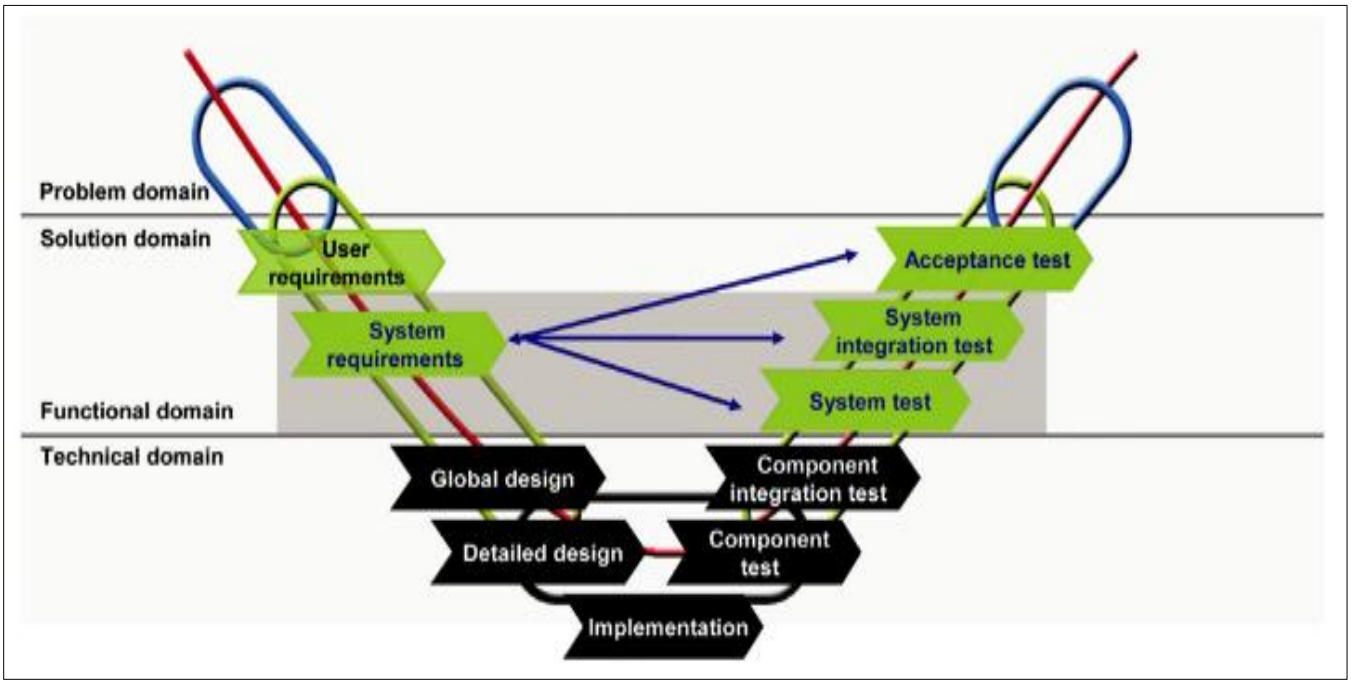

Fig 5: Position of informal and formal MBT in the V-model

5. Model-based Testing

Simply put, a model of software is a depiction of its behavior. Behavior can be described in terms of the input sequences accepted by the system, the actions, conditions, and output logic, or the flow of data through the application's modules and routines. In order for a model to be useful for groups of testers and for multiple testing tasks, it needs to be taken out of the mind of those who understand what the software is supposed to accomplish and written down in an easily understandable form. It is also generally preferable that a model be as formal as it is practical. With these properties, the model becomes a shareable, reusable, precise description of the system under test. There are numerous such models, and each describes different aspects of software behavior. For example, control flow, data flow, and program dependency graphs express how the implementation behaves by representing its source code structure. Decision tables and state machines, on the other hand, are used to describe external so-called black box behavior. When we speak of MBT, the testing community today tends to think in terms of such black box models

Models in Software Testing

We cannot possibly talk in detail about all software models. Instead, we introduce a subset of models that have been useful for testing and point to some references for further reading. Examples of these are finite state machines, state charts, the unified modeling language (UML) and Markov chains.

Finite State Machines

Consider a common testing scenario: a tester applies an input and then appraises the result. The tester then selects another input, depending on the prior result, and once again reappraises the next set of possible inputs. At any given time, a tester has a specific set of inputs to choose from. This set of inputs varies depending on the exact "state" of the software. This characteristic of software makes state-based models a logical fit for software testing: software is always in a specific state and the current state of the application governs what set of inputs testers can select from. If one accepts this description of software then a model that must be considered is the finite state machine. Finite state machines are applicable to any model that can be accurately described with a finite number (usually quite small) of specific states. 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. [Chow 1978] [10] was one of the earliest, generally available articles addressing the use of finite state models to design and test software components.

State Charts

State charts are an extension of finite state machines that specifically address modeling of complex or real-time systems. They provide a framework for specifying state machines in a hierarchy, where a single state can be "expanded" into another "lower-level" state machine. They also provide for concurrent state machines, a feature that has only a cumbersome equivalent in automata theory. In addition, the structure of state charts involves external conditions that affect whether a transition takes place from a particular state, which in many situations can reduce the size of the model being created. State charts are intuitively equivalent to the most powerful form of automata: the Turing machine.

However, state charts are more pragmatic while maintaining the same expressive capabilities. State charts are probably easier to read than finite state machines, but they are also nontrivial to work with and require some training upfront.

Unified Modeling Language

The unified modeling language or UML models have the same goal as any model but replace the graphical-style representation of state machines with the power of a structured language. UML is to models what C or Pascal are to programs – a way of describing very complicated behavior. UML can also include other types of models within it, so that finite state machines and state charts can become components of the larger UML framework.

You can read about UML in [Booch *et al*. 1998] [9]. UML has gained a lot of attention lately, with commercial tool support and industrial enthusiasm being two essential factors. An example commercial tool is Rational's Rose, and many public domain tools can be found and downloaded from the web. For more on Rational Rose and UML, try [Rational 1998] and [Quatrani 1998].

There is a recent surge in the work on UML-based testing. Examples of this are many, but for a sample refer to [Abdurazik & Offutt 2000] [1], [Basanieri & Bortolino 2000] [6], [Hartman *et al*. 2000], [Offutt & Abdurazik 1999], and [Skelton *et al*. 2000] [22].

Markov Chains

Markov chains are stochastic models that you can read about in [Kemeny & Snell 1976]. 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 [Avritzer & Larson 1993] [4] and [Agrawal & Whittaker 1993] [2]. Work related to measurement and test can be found in [Whittaker & Thomason 1994] and [Whittaker & Agrawal 1994], [Walton & Poore 2000 IST] [23], [Whittaker *et al*. 2000]. Other general work worth looking at include [Doerner & Gutjahr 2000] [14] and [Kouchakdjian & Fietkiewicz 2000].

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.

Other Models

There are other models that may be worth investigating. Examples include decision tables, decision trees, and program design languages. Read [Davis 1988] [13] for a comparative overview of a number of these models in addition to some that have been mentioned in this section.

6. Closing Thoughts

Advantages of MBT

Model-based techniques have substantial appeal. The first sign of potential are studies showing that testing a variety of applications has been met with success when MBT was employed.

The reported experiences seem to indicate that MBT is specifically tailored for small applications, embedded systems, user interfaces, and state-rich systems with reasonably complex data. If these claims are substantiated, we should see MBT applied to various software: embedded car, home appliance and medical device software, hardware controllers, phone systems, personal digital assistant applications, and small desktop tools and applets. Investigations into whether MBT techniques are suitable for programmable interfaces, web-based applications, and data-intensive systems have begun. Results are beginning to be reported (see [Jorgensen 2000] and [Paradkar 2000].

In addition to the many software contexts in which MBT seems to excel, there are qualities attractive to model-based approaches by mere virtue of employing a model. It is worthwhile to mention two in this introduction. First, a model serves as a unifying point of reference that all teams and individuals involved in the development process can share, reuse, and benefit from. Second, most popular models have a substantial and rich theoretical background that makes numerous tasks such as generating large suites of test cases easy to automate. For example, graph theory readily solves automated test generation for finite state machine models [Robinson 1999 TCS], and stochastic processes gave birth to the popular Markov chain models that can aid in estimating software reliability [Whittaker & Thomason 1994].

Finally, using the right model constitutes the necessary basis for a framework for automating test generation and verification. By using a model that encapsulates the requirements of the system under test, test results can be

evaluated based on matches and deviations from what the model specifies and what the software actually does.

Difficulties and Drawbacks of MBT

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 sometimes-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 generation, and achieving coverage criteria. This topic will be addressed below.

7. Conclusion & Future Scope

The test models gave the testers a better overview than the textual specification and were very useful in communicating with functional designers. However, knowledge is needed to understand the test models. For this reason the test models used are not considered to be business user friendly.

Good software testers cannot avoid models. They construct mental models whenever they test an application. They learn, e.g., that a particular API call needs to be followed by certain other calls in order to be effective. Thus, they update their mental model of the application and apply new tests according to the model. MBT calls for explicit definition of the model, either in advance or throughout (via minor updates) the testing endeavor. 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.

Unfortunately it is not possible yet to make a difference in the test generation for high or low risks requirements. Currently test cases are generated from the requirements as if they all have the same risk level. When executing test cases manually, this means that the test cases have to be sorted out by hand. In general MBT can certainly help to improve the test process and test results. Savings in time and money are expected once the people are trained in modeling and initial test models are designed.

8. References

1. Abdurazik, Offutt, Aynur Abdurazik, Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 00), York, UK. 2000.
2. Agrawal, Whittaker K, Agrawal, James A. Whittaker. Experiences in applying statistical testing to a real-time, embedded software system. Proceedings of the Pacific Northwest Software Quality Conference. 1993.
3. Apfelbaum, Doyle, Larry Apfelbaum, Doyle J. Model-based testing. Proceedings of the 10th International Software Quality Week (QW 97). 1997.
4. Avritzer, Larson, Alberto Avritzer, Brian Larson. Load testing software using deterministic state testing. Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA 1993), ACM, Cambridge, MA, USA. 1993, 82-88.
5. Avritzer, Weyuker, Alberto Avritzer, Elaine J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. IEEE Transactions on Software Engineering. 1995; 21(9):705-715.
6. Basanieri, Bortolino, Basanieri F, Bertolino A. A practical approach to UML-based derivation of integration tests. Proceedings of the 4th International Software Quality Week Europe (QWE 2000), Brussels, Belgium. 2000.
7. Beizer. Boris Beizer. Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley. 1995.
8. Binder, Robert Binder V. Testing object-oriented systems. Addison-Wesley, Reading, MA, USA. 2000.
9. Booch *et al.* Grady Booch, James Rumbaugh, Ivar Jacobson. The unified modeling language. Documentation Set, Version 1.3, Rational Software, Cupertino, CA, USA. 1998.
10. Chow, Tsun Chow S. Testing design modeled by finite-state machines. IEEE Transactions on Software Engineering. 1978; 4(3):178-187.
11. Clarke, James Clarke M. Automated test generation from a behavioral model. Proceedings of the 11th International Software Quality Week (QW 98). 1998.
12. Dalal *et al*. Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM. Model-based testing of a highly programmable system. Proceedings of the 1998 International Symposium on Software Reliability Engineering (ISSRE 98), Computer Society Press. 1998, 174-178.
13. Davis, Alan Davis M. A comparison of techniques for the specification of external system behavior. Communications of the ACM. 1988; 31(9):1098-1113.

14. Doerner, Gutjahr, Doerner K, Gutjahr WJ. Representation and optimization of software usage models with non-Markovian state transitions. Information and Software Technology. 2000; 42(12):815-824.
15. Duncan, Hutchinson, Duncan AG, Hutchinson JS. Using attributed grammars to test designs and implementations. Proceedings of the 5th International Conference on Software Engineering (ICSE 1981), San Diego. 1981.
16. El-Far, Ibrahim El-Far K. Automated Construction of Software Behavior Models. Master's Thesis, Florida Institute of Technology. 1999.
17. Fujiwara *et al*. Susumu Fujiwara, Gregor Bochmann v, Ferhat Khendek, Mokhtar Amalou, Abderrazak Ghedamsi. Test selection based on finite state models. IEEE Transactions on Software Engineering. 1991; 17(6):591-603.
18. Gronau *et al*. Ilan Gronau, Alan Hartman, Andrei Kirshin, Kenneth Nagin, Sergey Olvovsky. A methodology and architecture for automated software testing. IBM Research Laboratory in Haifa Technical Report, MATAM Advanced Technology Center, Haifa 31905, Israel, [Gross & Yellen 1998] Jonathan Gross and Jay Yellen. Graph theory and its applications. CRC, Boca Raton, FL, USA. 1998-2000.
19. Harel, Harel D, State charts. a visual formalism for complex systems. Science of Computer Programming. 1987; 8(3):231-274.
20. Hartman *et al*. Hartmann J, Imoberdorf C, Meisinger M. UML-based integration testing. Proceedings of the 2000 International Symposium on Software Testing and Analysis. 2000.
21. Hong. Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo Hwan Bae, Hasan Ural. A Test Sequence Selection Method for State charts. The Journal of Software Testing, Verification & Reliability. 2000; 10(4):203-227.
22. Skelton *et al*. Skelton G, Steenkamp A, Burge C. Utilizing UML diagrams for integration testing of object-oriented software. Software Quality Professional. 2000.
23. Walton, Poore, Gwendolyn Walton H, Jesse Poore H. Generating transition probabilities to support model-based software testing. Software: Practice and Experience. 2000; 30(10):1095-1106.
24. Zhu *et al*. Hong Zhu, Patrick Hall AV, John HR. May. Software unit test coverage and adequacy. ACM Computing Surveys. 1997; 29(4):366-427.