Enjoying the Perks of Model-Based Testing

Ibrahim K. El-Far, *Florida Institute of Technology*

Abstract

Software testing demands the use of some model to guide such test tasks as selecting test inputs, validating the adequacy of tests, and gaining insight into test effectiveness. Most testers gradually build a mental model of the system under test, which would enable them to further understand and better test its many functions. Explicit models, being formal and precise representations of a tester's perception of a program, are excellent shareable, reusable vehicles of communication between and among testers and other teams and of automation for many tasks that are normally tedious and labor-intensive. Model-based testing offers advantages like automating test generation and providing a basis for statistically estimating product quality. These perks can be enjoyed provided the right models are used, the proper resources acquired, and adequate training undergone. Model-based testing is not without difficulties, and knowing what they are and whether they can be avoided and how is key to reaping the most out of it.

Keywords

Model-based testing, finite state models, training, testing resources, choosing models, application types, pros and cons

1. Introduction

Model-based testing or MBT is deep rooted in the worlds of hardware and phone switches, and its value is well appreciated there. MBT carries a lot of promise in the software realm, and the published research, both in academic and industrial settings, is helping us better understand this paradigm, but its worth is yet to be validated. Do the merits outweigh the costs? In what contexts is this approach preferred? What models are more appropriate for a specific application type?

The research successes have led many companies into employing MBT, and the reported results have been encouraging but not conclusive. More collaborative research needs to be done before significant results are collected and shared with those who are interested and who can benefit in the industry. There is certainly enough incentive to justify a closer look by every organization that envisions modeling as an essential part of its technical future. Such investigation would be a chance to understand more about modeling and to validate the worth of model-based testing.

1.1 About This Paper

This paper is a quick guide into the pros and cons of model-based testing: what's in it for you, what it is going to cost your organization, and what sort of resources and training you would need to get started. You will glance at some of the important issues in MBT like how to choose a model, and how to handle some of the more common problems in this field.

This paper is not a tutorial, however, nor is it a survey of earlier and current work, although I will try to point out references from which you can pick up and read further. This paper is a summary of some of the lessons I learned doing model-based testing, observing groups practicing it, and teaching it to inexperienced testers over the past few years. Consequently, despite my best intentions, I am sure it suffers from the biases of my own experiences in finite state machine model-based testing.

Lastly, I would like to point that this is part of a work in progress. I am constantly seeking opinions and experiences on various model-based testing affairs. My lists on advantages, requirements, and problems are always being updated. Most importantly, I am currently investigating the usefulness of various models in different software contexts and the manner in which models can be combined to yield maximal return on investment.

1.2 Organization

I start out with an introduction that presents a brief argument as to the need of models in testing and establishes some of the benefits to basing testing activities on an explicit, formal model, but also points out that there is never a free lunch. This is followed by a quick overview of model-based testing activities, which, again is not intended to tutor you in MBT. This overview focuses on tips that help formulate a mental model of the system under test prior to constructing an explicit representations, as well as general comments on building models.

The remainder of the paper stems almost exclusively from personal experience. I briefly go through some of what I regard as key MBT advantages. Next, I walk you through some examples in which certain models have been exhibited to fit testing needs better than others. Finally, I talk about two challenges that, although may be common to other paradigms, can severely manifest themselves in model-based testing and can overturn potentially successful MBT effort into utter failure. I close with a short section that contains bibliographical pointers and a list of the references used in this paper.

1.3 Models: From a Tester's Head Into the World of Shareable Resources

Software testing demands the use of some model to guide its various tasks, for example selecting tests, validating the tests are adequate, and making the decision to stop testing. More often that not, such a model is implicit, existing only in the head of a human tester, who probably starts out by exploring the functions of the system and applying inputs in an ad hoc fashion (or perhaps with certain objectives that are typically vague). Over a period of time, a tester would build a mental model that encapsulates the behavior of the system under test and that reflects a better understanding of the application's capabilities. This evolving mental model contributes to more effective testing, and, it is combined with the tester's experience in testing, in her previous dealings with the problem domain of the software, and her other related knowledge and know-how to make for a first class tester.

Alas, the wonders of good mental models are imprisoned in the minds of good testers, and whatever does get communicated to the outside world is scrambled by the limitations and ambiguities of natural language. However, certain aspects of these mental models can be captured, formalized, and written down, according to some standard or theory. When a model becomes explicit, whatever information it embodies becomes shareable and reusable. Those who build an explicit model of software behavior would be performing the most important step of what has come to be known as model-based testing or MBT.

1.4 Software Models and Today's Industry

The software industry is realizing the significance of models in understanding requirements, building designs, and guiding other software development activities. Models are not a new concept in philosophy or science and they are certainly not new in software; certain types of models have been built almost since the inception of computing. So what is behind models and, consequently, MBT coming under the industrial spotlight? There can be many speculations, but, without doubt, the advent of object oriented technology and the increasing popularity of modeling tools and languages such as UML have played a big role. In addition, the interest in quality is renewing everyday, and there are few who doubt that the quality of a software product will become the dominant success factor in a near future. In an industry always looking for ways to make things better, the merits of model-based approaches are appreciated for the new light they shed on what can be done to improve the quality of a program.

There are many types of models that can be adequate for describing the various aspects of software behavior: decision tables, finite state machines and variations, Markov chains, statecharts, Petri-nets, entity-relationship diagrams, object models, and many more. There are many who have talked about models in the contexts of requirement analysis, design, specifications, and testing. I always like to start with Alan Davis' "A comparison of techniques for the specification of external system behavior" (1988) in which he describes and compares various approaches to specifying software, and I also like to read Sommerville and Sawyer's parts of "Requirements Engineering: A Good Practice" (1997) on hints on choosing a model. It is interesting to note that, when looking at the testing literature, it seems that state machines and similar models like Markov chains are arguably the most popular software representations in the world of testing.

1.5 Why All the Commotion About MBT?

MBT zealots will tell you that using it will bring so many benefits that it is hard to imagine why anyone would test any other way and why MBT is not the industry's mainstream. Indeed that are many advantages to using a model: a high degree of automation, the ability to generate high volumes of non-repetitive useful tests, means to evaluate regression test suites, and the possibility of estimating a number of statistical measures of the software's quality. There are also immeasurable benefits of having a model as a way to communicate ideas and as a living document of the software under test and associated testing activities. Of course, not all the models offer the same advantages to the same extent, and some may be harder to work with than others depending on a variety of factors, some of which will be exposed in this paper. However, given the right conditions, many have reported success when using MBT, especially on phone systems, controller software, and all sorts of graphical user interfaces. It such reports that have led to an increasingly better understanding of models in testing and created interest in this blooming area.

1.6 What is in the Fine Print?

All testers know, I hope, that there are no silver bullet methods: different strategies may work differently along the spectrum of application types. Similarly, different models lend themselves to varying extents to different application types. Trying to use an inadequate model to describe some sort of behavior can become tedious, even anti-productive. On several occasions, I have seen groups trying to model systems with state models for two weeks, when using a grammar would have been no less powerful and accomplished in, say, one or two days. What I am trying to say is that, in order to reap all the promised benefits of a model, it needs to be used only when the application type and other circumstances are permitting (organizational, project, and application considerations). Furthermore, you have to remember that it may be necessary to combine several types of models in order to satisfy various testing-objectives. The all-too-popular view of the one-model solution in MBT is ultimately as useful and successful as the one- strategy solution is in testing.

That seems reasonable enough, you may say, but what's the catch? How much is it going to cost you? What resources are you going to have to ask for or buy? Are there common difficulties and there ways around them? The fine print on the MBT package is that the many benefits of modeling do not usually come cheap and may require tools, skills, and resources for them to be attained. Moreover, there are problems and difficulties that are inherent to using models, as there are problems and difficulties inherent to using many other rigorous approaches to testing. Without training MBT may become too ineffective to be profitable. Also, without the tools, some MBT tasks may not even be possible. Some problems can be solved, but other hurdles need to be worked around if any progress is to be made. The good news is that knowing what is there to be gained and being aware of the limitations and obstacles will most likely maximize the chances of enjoying MBT to the fullest extent possible. At the very least, you will be able to make a decision as to whether MBT fits your specific needs and purposes.

2. A Model-Based Testing Jumpstart

My view of testing is that it is not merely a single-tasked phase in which bugs are uncovered and fixed. Testing is a complete process that is, in many ways, analogous to though not necessarily as rigorous or involved as the software engineering process. First testers build a progressively refined mental model of the system under test by understanding the various aspects of its features. Next, as the mental model is evolving, testers try to express it in precise terms of one or more models. This is in order to establish better communication among testers and in order to make sure that the testing team has a consistent view of what the feature being tested is all about.

So as the mental model is growing, and as various models are being built in an incremental fashion, traditional testing tasks can be commenced. Based on test adequacy criteria, stopping criteria, and the structure of the software representation (model), tests are automatically generated. Test suites are then applied to the software using some simulation mechanism, and the test results are observed and evaluated. Finally, the test failure information is used to determine whether more testing is needed, and to estimate the quality of the product as it.

Meanwhile, testers are still exploring the application and studying it, perhaps also updating the models as well. Based on the results of applied test suites, testers may decide to improve and refine the model, or, if they decide that the feature under test has been thoroughly tested, they can move on to other features, and build models for those in a similar fashion.

The figure below is a simplified depiction of all these activities just described. It is important to note that, although the figure looks like a waterfall model of testing, the reality is that the results of every task from building the mental models to evaluating test results and making subsequent decisions are used to improve and revisit other performed tasks.

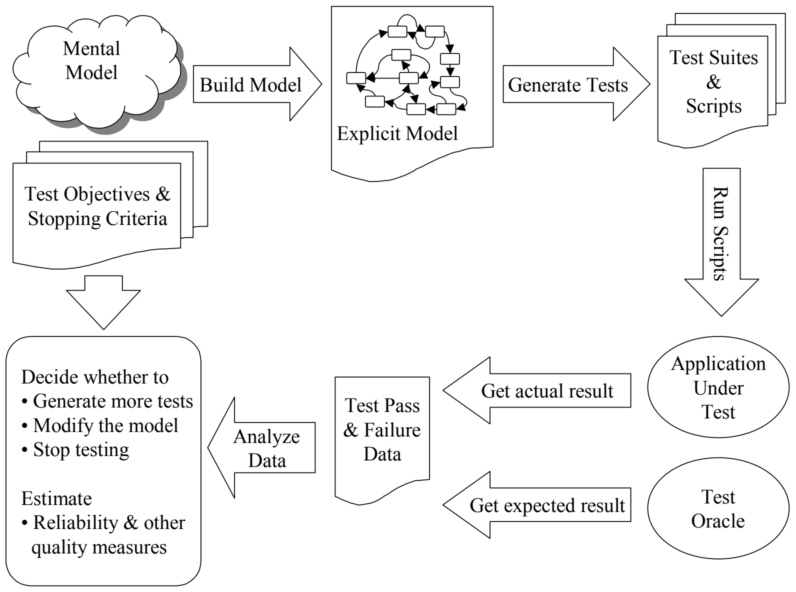

Figure 1 A Simplified View of Some Typical Model-Based Testing Tasks

In the remainder of this section, I talk about (1) building an understanding of a piece of software and (2) about constructing models, the two most crucial and error-prone tasks in model-based testing. They are the most crucial because they produce the model upon the quality of which rests the success of all other tasks. They are the most error prone because they are most human-dependent: testers' skills, experience, training, and understanding all play a big role. I disregard the other testing tasks because I feel that many others have talked about them and because of the limited scope and size of this paper.

2.1 Developing and Refining a Mental Model

The requirement common to most styles of testing is a well-developed understanding of what the software accomplishes, and MBT is no different. Forming a mental representation of the system's functionality is a prerequisite to building models. This is a nontrivial task as most systems today typically have convoluted interfaces and complex functionality. Moreover, software is deployed within gigantic operating systems among a clutter of other applications, dynamically linked libraries, and file systems all interacting with and/or affecting it in some manner. To develop an understanding of an application, therefore, testers need to learn about both the software and its environment.

Here, experience is a valuable asset that can accelerate the formation of a good mental model, but it can also be a double-edged sword, where strong biases may turn attention away from circumstantially significant issues. On the other hand, without much experience, novices may take considerable time to understand the program they intend to test. Here are some tips on the kind of activities that may be performed to improve program understanding.

1. *Determine the components/features that need to be tested based on test objectives.* No model is ideal to completely describe a complex or large system. Determining what to model for testing is a first step in keeping MBT manageable.

2. *Start exploring target areas in the system.* If development has already started, acquiring and exploring the most recent builds with the intent of learning about functionality and perhaps causing failures is a valuable exercise toward constructing a mental model of the software.

3. *Gather relevant, useful documentation.* Like most testers, model-based testers need to learn as much as possible about the system. Reviewing requirements, use cases, specifications, miscellaneous design documents, user manuals, and whatever documentation is available are indispensable for clarifying ambiguities about what the software should do and how it does it. However, if the documents are long, vague, or out of date, you may not want to spend too much effort: find alternative sources of the information you seek.

4. *Establish communication with requirements, design, and development teams if possible.* Talking things over with other teams on the project can save a lot of time and effort, particularly when it comes to choosing and building a model. There are companies that practice building several types of models during requirements and design. Why build a model from scratch if it can be reused or adapted for testing purposes, thus saving significant time and other resources? Moreover, many ambiguities in natural language and/or formal documents can be better resolved by direct contact rather than reading stacks of reports and documents.

5. *Identify the users of the system.* Each entity that either supplies or consumes system data, or affects the system in some manner needs to be noted. Consider user interfaces; keyboard and mouse input; the operating system kernel, network, file, and other system calls; files, databases and other external data stores; programmable interfaces that are either offered or used by the system. This identification is a first step to study events and sequences thereof, which would add to testers' ability to diagnose unexpected test outcomes.

6. *Enumerate the inputs and outputs of each user.* In some contexts, this may sound like an overwhelming task, all users considered, and it is tedious to perform manually. However, dividing up the work according to user, component, or feature greatly reduces the tediousness. Additionally, there are commercially available tools that alleviate much of the required labor by automatically detecting user controls in a graphical user interface and exported functions in programmable interfaces.

7. *Study the domains of each input.* In order to generate useful tests in later stages, real, meaningful values for inputs need to be produced. An examination of boundary, illegal, and normal/expected values for each input needs to be performed. If the input is a function call, then a similar analysis is required for the return value and each of the parameters. Subsequently, intelligent abstractions of inputs can be made to simplify the modeling process. Inputs that can be simulated in the same manner with identical domains and risk may sometimes be abstracted as one. It may also be easier to do the reverse: calls of the same function with non-equivalent parameters may be considered as different inputs.

8. *Document input applicability information.* To generate useful tests, the model needs to include information about the conditions that govern whether the user can apply an input. For example, the human user cannot press a button in a particular window if that window is not open and active.

9. *Document conditions under which responses occur.* A response of the system is an output to one its users or a change in its internal data that affects its behavior at some point in the future. The conditions under which inputs cause certain responses need to be studied. This not only helps testers evaluate test results, but also design tests that intentionally cause particular responses.

10. *Study the sequences of inputs that need to be modeled.* This vital activity leads straight to model building and is where most of the misconceptions about the system are discovered (or worse, formed). The following questions need to be answered. Are all inputs applicable at all times? Under what circumstances does the system expect or accept certain input? In what order is the system required to handle related inputs? What are the conditions for input sequences to produce particular outputs?

11. *Understand the structure and semantics of external data stores.* This activity is especially important when the system keeps information in large files or relational databases. Knowing what the data looks like and what it means allows weak and risky areas to be exposed to analysis. This can help build models that generate tests to cause external data to be corrupted or tests that trigger faults in the system with awkward combinations of data and input values.

12. *Understand internal data interactions and computation.* As with the previous activity, this adds to the modeler's understanding of the software under test and consequently the model's capabilities of generating bug-revealing test data. Internal data flow among different components is vital to building high-level models of the system. Internal computations that are primarily risky arithmetic, like division or high-precision floating-point operations, are normally fertile ground for faults.

13. *Maintain one living document: the model.* Opinions vary on this, but unless required by organizational regulations, there is little reason to create a document of all relevant information. Maintaining a set of pointers to all documents that contain the needed information is sufficient. It is also quite reasonable to annotate the model with comments on modeling rationale and observations on the program especially if there is a lack of proper documentation. In the end, that is what models are all about: to express an understanding of software behavior.

2.2 Building the Model

Building a model is a creative effort; it involves the skills, experience, and imagination of the human tester, and, in that sense, it is much like writing code. Code is a formal precise artifact, and so are models. However, there are countless ways to write code that does a specific job. Similarly, there are many (literally an infinity of) models that describe the same exact program behavior. The choice of a model in MBT is like the selection of a programming language to implement some system: the decision on what model to use is critical to the productivity and efficiency of all model-based tasks in the testing process and is discussed later.

What should concern you while building a model is this. How well a model expresses program behavior corresponds, in development, to how well a program complies with its engineers' specifications and its customers' expectations; a good, expressive model leads to convincing, insightful results of the testing effort. Unfortunately, most really good tips are specific to particular models, but the following guidelines are always useful to keep in mind about your model:

1. The model needs to be as brief as possible without sacrificing information
2. The model should not contain information that are not pertinent for your objectives
3. The model should exclude pertinent information that is deemed redundant or useless
4. The model needs to be as readable as possible, even at the expense of brevity
5. The model needs to be left open for changes and additions, if at all possible
6. The model writing needs to consider all test implementation details even if the model does not

Here's a small example. Consider the two finite state models in the figure below. They are two attempts at describing the behavior of a simple phone system. Observe the following:

1. Both models seem to be minimal in size and neither seems to contain redundant or useless information about a phone receiver.

2. Model B has more non-redundant states than Model A. In other words, there is *more information* expressed in Model B (although we cannot conclude, based only on this observation that the additional information is necessary or correct).
3. Model A allows for the following sequence: the user picks up the phone, calls a busy party, and then the party picks up. Model B does not allow this: the user picks up the phone, calls a busy party, and then the only thing left to be done is to hang up (from the busy state). Not only Model A has less detail, but it *describes behavior that the phone should not exhibit*. This can lead to generating "legal" inputs for which the phone will "fail."

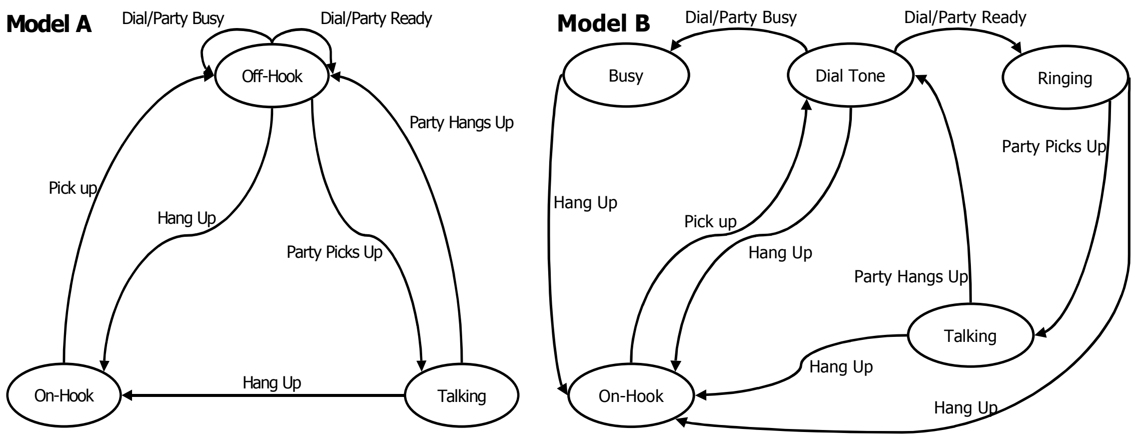

Figure 2 Two Finite State Models of a Simple Phone System

3. A Brief Discussion of Advantages

Here are some immediate returns on a serious investment in model-based testing:

1. *Modeling is a precise communication tool among teams. Certain models are sometimes excellent vehicles of presentation to non-technical management (a model is worth a thousand words).*

2. *The model is a living, accurate document of the aspects of the system it expresses.*

3. *Intensive Automation and Tests of a Different Nature.* Using models one can automatically generate large suites of tests. Observe how in either of the models shown above, by traversing the graphical structure of the model, one can general virtually infinite number of non-repetitive tests (or sequences of inputs). Individual tests in a suite can be lengthy which may suite the needs of a testing team after failures that reveal themselves after a long period of time (see below). Finally, individual tests can be based on what a user is likely to do, with the aid of, say, an operational profile, or they can be purposefully complicated, odd sequences of inputs. What is particularly appealing about using a model is that you only need to maintain a minimal number of test scripts (for example, scripts needed to simulate individual inputs, and scripts needed to force the system into a particular starting state of the model). All one needs to store if the test generation method (exact adequacy criteria), the stopping criteria, and, if applicable, the random seed, and the random number generator, because that is all that is needed to deterministically know the tests that have been selected.

4. *Failures exposed by MBT.* Because of the kind of tests that can be generated using models (see previous point), MBT is likely to expose failures that are caused by a weird combination of inputs; that are revealed only with time (memory leaks); and that are revealed by exercising different combinations of system variables (whatever is described in the model).

5. *Coverage.* Various forms of coverage are used to either evaluate test progress or evaluate the adequacy of the generated tests. Coverage can also be expressed in terms of a model, which has

no direct precise known relationship with any other types of coverage (statement, definition-use paths, requirements, etc…). Model coverage is therefore yet another heuristic that provides insight into the thoroughness and effectiveness of the testing effort, especially when testing reveals no failures.

6. *Metrics*. Certain models have inherent features that make computing metrics like mean time to failure and reliability a bit easier (for example, Markov chains).

8. *Merits of Modeling with the Purposed of Testing*. Modeling an application is a rigorous process during which testers will develop a better understanding of the software under test, and, while trying to build a model, they will uncover more faults that model-based testing itself may not uncover.

9. *Models of various versions of a product can be used to monitor its growth/evolution*.

10. *Models can help is regression test efforts.* Because of the nature of models, testing can be steered to test the areas of the program that have been changed with non-repetitive tests (tests that have not been tried before). However, in the case that testers feel the urge/obligation to make sure that certain tests pass, then there is no need to maintain suites of regression suites for the same reason that there is no need to maintain test scripts (stated above). In addition, if testers do maintain test suites and their scripts, it is possible to use models to validate these tests cheaply, that is to make sure that, after the fix and changes in the model, these tests should run fine.

11. *Modeling facilitates more rigorous verification*. Because of the formal and precise nature of modeling, such activities as program proof, precondition analysis, model checking, and other forms of formal verification that increase confidence in software become less tedious.

4. Does the Model Fit the Needs

In isolation of all human factors like skills, education, and experience, and all project circumstances like the managerial support, resource availability, and time allocated for testing activities, there are two major issues to be considered when choosing a model for model-based testing. The first is the application type, or the type of feature that is about to be tested, and the other is the body of knowledge associated with the model type proposed to represent that feature.

4.1.1 Application Type

No large-scale studies have been made to verify the claims of the supports of any particular models. All we have is some intuitive observations from various experiences and insight into why we have encountered considerable or limited success.

Example 1. To model HTML files for a web browser or mathematical expressions for a scientific calculator, a grammar is probably the quickest and easiest to build, since grammars are typically used to describe such languages. In addition, for decent browsers and calculators, such grammars would already be available, especially since they would be needed to build HTML and mathematical expression parsers.

Example 2. Phone systems are typically state rich and many faults in such systems are detected by load testing for a prolonged period, making state machines an ideal solution for those purposes. This has been validated by several experience reports in the past.

Example 3. Parallel systems are expected by definition to have several components that run concurrently, perhaps each with a different model. If the individual components can be modeled using state machines, then statecharts are a reasonable solution for a couple of reasons. A statechart can be in more than one state at a time, and a statechart state can itself be a statechart. Otherwise, if state models are inappropriate, ways have to be improvised to combine different models from different components to synchronize them and satisfy other possible time constraints. This area is one gaping hole in MBT theory.

Example 4. If a system can be in one of very few states but transitions between states are governed by several external conditions in addition to inputs, then a model more potent than a simple finite state machine is needed; statecharts may be appropriate, because transitions in a statecharts are governed by a triggering input as well as an external condition.

Example 5. If a system can be modeled with a finite state machine, but there is intention of statistical analysis of failure data, then Markov chains may be more suitable than state machines.

Example 6. If operational profiles are to guide test generation, Markov chains are good candidates (assuming the system can be modeled using state machines).

Example 7. If we are interested in determining whether the system accepts sequences of inputs in a particular structure, as in the case of compilers accepting programs written in the language it is supposed to compile, then grammars are usually a better choice (similar to example 1).

Example 8. Grammars may also be a preferred model when testing network packet processors and other protocol-based applications (we try to look at structure and semantics of packets, rather than, say, the sequence of packet contents, which is not too complicated).

Example 9. There are times at which the goal is to test whether software behaves correctly for all combinations of values for a certain number of inputs. A tabular model that encodes these combinations would be particularly useful for this.

Example 10. If there is need to represent conditions under which inputs cause a particular response, but state machines are too awkward, decision tables fit those particular needs better.

4.1.2 BODY OF KNOWLEDGE RELATED TO THE MODEL

All things considered, it is how much we know about a model together with what we know about the application domain that usually determines what alternative we pick. Consider the case of state machines. Automata theory gives us a classification of state machines and the types of languages they can represent. This means that by understanding what an application does, basic automata theory can recommend a model or, conversely, tell us how much of the system we can model using a particular state machine. Automata theory gives means of determining whether different models are equivalent. For example, statecharts are not equivalent to finite state machines but it is to some other form of state machine. In addition different classes of state machines are equivalent to corresponding classes of grammars, in terms of expressive power. Moreover, finite state machines can be looked at as directed graphs, structurally speaking. Since tests generated based on a model are simply paths in that graph, graph theory can help by supplying traversal algorithms and achieving graph-structure coverage criteria for testing. Finally, state machines have been used in computer hardware testing for a long time, and there is a sizeable associated body of knowledge.

Grammars are part of the same fundamental automata theory. We know as much about grammars as about state machines, when it comes to what software can be modeled with what kind of grammars. As far as test generation, the simplest form of a test generator is a reverse compiler that produces structurally and semantically appropriate sequences of inputs (tokens) instead of parsing them. Since compiler tools are widely available, creating a grammar-based test generator is not difficult. Unfortunately, there is nothing similar to graph theory that can help in guided testing and in setting and achieving coverage criteria in the world of grammars.

5. Resources, Training and Other Issues

5.1 Skills

Skills of people who will deal with the model are another consideration. Testers who build the model need to have been educated in the basics of the underlying theories and trained in modeling practices. For finite state machines, this includes elements of automata theory, graph theory, and methods to build state machines. Add to that statistics and stochastic processes in the case of Markov chains, which can be looked at as probabilistic state machines.

5.2 Audience

Highly formal models are probably adequate if its audience is limited to the technically oriented, who are expected to create and read a model, review it, and maintain it. If the audience encompasses managers and other personnel of various backgrounds who are expected to review the model at some point, then something in between the optimal-for-testing and best-for-human-understanding is more appropriate. Grammars and state machines are generally not very readable, especially when compared to tabular representations. The good news is that there are typically many ways to convert one model into another, especially when the underlying theory has been well established.

5.3 Tools

The skills of model-based testers are not enough to lead a successful MBT endeavor. In the absence of tools that support various MBT activities, the costs testing teams will incur may not be justified from a bug-count perspective even over the long run. Organizations that wish to test using a model that has no tool support should consider developing their own framework and corresponding tools to build, manage, and maintain models: it is not difficult and the cost can be easily justified. Examples of tools include: tools that examine the application to list its inputs and investigate some input applicability information, tools to build, manipulate and maintain models, test generators, test script managers and simulators, automated oracles (this is not something you can buy off the shelf, unfortunately), and tools to gather and compute metrics.

5.4 Existing Models

An organization already employing models for clarify requirements and building designs is better off employing or adapting those models for testing purposes. This would make the model a truly shared representation of the software, shrinking the chances for discrepancies between the views of testers and everyone else of what the software is built to do. A model used for design may not be appropriate for testing, and so projects should try to avoid this pitfall.

5.5 Process and Circumstantial Delays

An important matter is the point in time at which testing activities commence. If testing starts early, models can also be built early. Normally, models would have a high level of abstraction at first. Then, as testing finds no more bugs within reasonable time, and as fixes are introduced to the software, the model is modified, with progressively more detail. When new features are introduced for test, appropriate information can be modeled. When features are not to be tested anymore, relevant details are removed. Unfortunately, real-world schedules do not always allow for an early start, and model-based testers end up with the almost-prohibitive task of modeling parts of a truly large and complex system.

5.6 Short Development Cycles

There are also factors specific to environments of short development cycles. Model-based is generally time consuming, so it may not immediately pay off on a typical project. This is not acceptable when there is a new release every other week, so more versatile techniques are needed. MBT can be used for testing after the releases have achieved a certain degree of stability in features and delivery dates.

6. Two Practical Challenges

6.1 State Space Explosion

Every work that addresses building, maintaining, reviewing, checking, and testing using models mentions state space explosion. When the model is not state-based, the problem reveals manifests itself through an exponentially growing model size of the software. Reading Weyuker's "Testing Component Based Software: A Cautionary Tale" (1998) on her experiences in testing with large models is a pleasant and informative exposure of state space explosion among other issues.

State explosion cannot be avoided: you will face it if you do MBT one way or the other: the number of states will be overwhelming, the number of grammatical rules will become beyond manageable, the number of variable-value combinations will become prohibitively large. There are few ways of working around this. First, you can try to model that feature using one or more other types of models; some models tend to express certain program behavior more compactly (a good example is grammars versus state machines in modeling HTML or mathematical expressions mentioned earlier).

Second, if no other models seem appropriate or you cannot afford to use another model for any number of reasons, try abstracting complex information into simple representations without sacrificing critical knowledge about the program's behavior. For example, there is no need to represent all floating numbers that constitute possible values for the argument of a square root function. Abstracting those into three categories (explicitly partitioning the possible values) negative numbers, zero, and positive numbers would greatly simplify things. For another example, a dialog box that requires the entry of multiple fields of data followed by the OK button could be modeled as two abstract inputs: "enter valid data" and "enter invalid data." Abstraction by definition causes loss of model information, and it obviously can be misused. But when used wisely, it can retain the important information and ignore superfluous or irrelevant information without loss of effectiveness.

Third, you can try exclusion. Exclusion means simply dropping information without bothering to abstract it. This is mostly done when decomposing a set of software behaviors into multiple models. Each model will contain certain information and exclude other pieces of information, which would be represented in another model.

State explosion represents a serious challenge to the model-based tester; the use of multiple small models, abstraction, inclusion, and alternative types of models are only a few ways to work around it and perhaps avoid it altogether.

6.2 Oracles and Automation

Automated mechanisms that evaluate test results (called oracles) are crucial to model-based testing. Without them a major value of MBT, automation, is almost voided. Unfortunately, there is little work that specifically addresses the oracle problem, and so it remains without so much as guidelines for a solution. There are, however, heuristics that provide an alternative to a complete oracle. Examples of oracles used in practice include (1) competing products, (2) different versions of the same product in which the feature under test did not experience significant change, and (3) different implementations developed by the same vendor. In addition, there are practical tricks such as (4) replacing the requirement of correctness with that of plausibility: if the test outcome appears to be within a reasonable range from the correct result, then the test passes.

7. Where To Look for Further Reading

To the best of my knowledge there is no comprehensive work that surveys models and their use in testing, although there is a considerable body of literature around the subject. Here are some places that I think are useful place to start for people interested in MBT.

The easiest place to start is the model-based testing home page (http://www.model-based-testing.org), maintained by Harry Robinson since 2000. It contains links to papers that you can download online, as well as a list of offline articles and books, all of which capture a wide range of MBT work.

There are no books on model-based testing that I know of. A good book for all testers, but that particularly addresses issues pertinent to MBT, is Paul Jorgensen's "Software Testing: A Craftsman's Approach" (1995). He discusses a number of white box models, and makes reference to black box models like decision tables. He has an informative concise introduction those elements of graph theory necessary for working with control and data flow models as well as state machines and other models that have a graphical representation.

"Model-Based Testing" is an entry to the field that James Whittaker and I wrote early in 2001, and it is rich in references on a wide variety of MBT work. In addition, Alexandre Petrenko has published an excellent annotated bibliography on testing with finite state models (2000).

Finally what follows is a sample of various reports on model-based testing experiences and novel work. The choice of papers is biased in that it is mostly from the academic literature and they predominantly involve finite state models, but they are representative much of the corresponding MBT work in the industry.

Consider the works of Rosaria and Robinson (2000) on testing graphical user interfaces in "Applying Models in Your Testing Process"; Agrawal and Whittaker (1993) on testing embedded controller software in "Experiencing in Applying Statistical Testing to a Real-Time, Embedded Software System"; and Avritzer and Larson on testing phone systems in "Load Testing Software Using Deterministic State Testing". For some enjoyable reading on testing with finite state machines try Robinson's papers in 1999: "Finite-State Model-Based Testing on a Shoestring" and "Graph Theory Techniques in Model-based Testing".

Work on grammar-based testing can be found in Duncan and Hutchinson's "Using Attributed Grammars to Test Designs and Implementations" (1981), and in Maurer's "Generating Test Data with Enhanced Context-Free Grammars" and "The Design and Implementation of a Grammar-based Data Generator" (1990 and 1992, respectively). Finally, work related to measurement and test can be found in Whittaker and Thomason's "A Markov Chain Model For Statistical Software Testing" (1994) and Walton and Poore's "Measuring Complexity and Coverage of Software Specifications" (2000).

Acknowledgements

Thanks are due to David Gelperin of Software Quality Engineering and Sam Guckenheimer of Rational Corporation for supplying lists of references in model-based testing and UML-based testing, respectively. Thanks are also due to James A. Whittaker for his academic supervision, financial support through the Florida Institute of Technology, and motivation and guidance in software testing. While I was working on this paper, I was funded in part by an IBM Center for Advanced Studies Fellowship that supported my other work on test cost and effectiveness for the year 2001. Ms. Florence E. Mottay of Florida Tech gave this paper a careful review that reshaped its organization and certain aspects of its style. Finally, over the last five years, I have benefited from discussion with friends, coworkers, and others on model-based testing issues: Alvin Flanegin, Anthony Berkemeyer, Nikhil Nilakantan, Herbert Thompson, Florence Mottay, Amit Singh, James Whittaker, Cem Kaner, Harry Robinson. If you like this work, thank those whom I just mentioned. If you do not approve then the blame, of course, is mine alone.

References

[Agrawal & Whittaker 1993] 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.

[Avritzer & Larson 1993] 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.

[Davis 1988] Alan M. Davis. A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9): 1098-1113, September 1988.

[Duncan & Hutchinson 1981] A.G. Duncan and J.S. Hutchinson. Using attributed grammars to test designs and implementations. *Proceedings of the 5th International Conference on Software Engineering (ICSE 1981)*, San Diego, March 1981.

[El-Far & Whittaker 2001] Ibrahim K. El-Far and James A. Whittaker. Model-based software testing. To appear in J.J. Marciniak, editor, *Encyclopedia of Software Engineering*, 2001.

[Jorgensen 1995] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC, August 1995.

[Maurer 1990] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, July 1990, pages 50-55.

[Maurer 1992] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice & Experience*, March 1992, pages 223-244.

[Petrenko 2000] Alexandre Petrenko. Fault model-driven test derivation from finite state models: annotated bibliography. *Proceedings of the Summer School MOVEP2000, Modeling and Verification of Parallel Processes*, Nantes, 2000 (*to appear in LNCS*).

[Robinson 1999 STAR] Harry Robinson. Finite state model-based testing on a shoestring. *Proceedings of the 1999 International Conference on Software Testing Analysis and Review (STARWEST 1999)*, Software Quality Engineering, San Jose, CA, USA, October 1999.

[Robinson 1999 TCS] Harry Robinson. Graph theory techniques in model-based testing. *Proceedings of the 16th International Conference and Exposition on Testing Computer Software (TCS 1999)*, Los Angeles, CA, USA, 1999.

[Robinson 2000] Harry Robinson. Intelligent Test Automation. *Software Testing & Quality Engineering Magazine*, September/October 2000.

[Rosaria & Robinson 2000] Steven Rosaria and Harry Robinson. Applying models in your testing process. *Information and Software Technology*, 42(12): 815-824, September 2000.

[Sommerville & Sawyer 1997] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice*. Wiley, April 1997.

[Whittaker & Thomason 1994] James A. Whittaker and Michael G. Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812-824, October 1994.

[Weyuker 1998] Elaine J. Weyuker. Testing component based software: a cautionary tale. *IEEE Software*, 15(5), September/October 1998.

About the Author

Ibrahim K. El-Far is a doctoral candidate in computer sciences at the Florida Institute of Technology under the academic supervision of professor James A. Whittaker. He has a Bachelor of Sciences and a Master of Sciences in Computer Science from the American University of Beirut, Beirut, Lebanon and the Florida Institute of Technology, Melbourne, Florida, USA, respectively, and he is a member of the Association for Computing Machinery. His interests are in investigating software models for testing, test automation and tools, adequacy criteria, test cost and effectiveness, and software testing education. In 2000, Ibrahim received a fellowship from the International Business Machines Center for Advanced Studies, Canada, supporting his research in approaches to interpreting, understanding, and analyzing software test effectiveness and efficiency. He has over four years of experience in model-based testing using finite state machines at the Center for Software Engineering Research at Florida Tech, where he has supervised the development of experimental model-based testing tools, advised model-based testing groups, and taught model-based testing in various formats to a variety of students. You can contact Ibrahim at ielfar@acm.org, visit http://www.testingresearch.com/ for more information, or write him at the Software Engineering Program, Computer Sciences Department, Florida Institute of Technology, 150 West University Boulevard, Melbourne, Florida 32901 USA.