Model-based Software Testing

Ibrahim K. El-Far and James A. Whittaker, *Florida Institute of Technology*

Abstract

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. There are a number of models of software in use today, a few of which make good models for testing. This paper introduces model-based testing and discusses its tasks in general terms with finite state models (arguably the most popular software models) as examples. In addition, advantages, difficulties, and shortcoming of various model-based approaches are concisely presented. 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, statecharts, unified modeling language, grammars, Markov chains, test automation, test case generation

1. Introduction

There is an abundance of testing styles in the discipline of software engineering today. Over the last few decades, many of these have come to be used and adopted by the industry as solutions to address the increasing demand for assuring software quality. During the last ten odd years, perhaps as an outcome of the popularization of object orientation and models in software engineering, there has been a growth in black box testing techniques that are collectively dubbed *model-based testing*. Model-based testing or MBT is a general term that signifies an approach that bases common testing tasks such as test case generation and test result evaluation (see [Jorgensen 1995] for some of these basic terms and concepts) on a model of the application under test.

MBT has as its roots applications in hardware testing, most notably telephone switches, and recently has spread to a wide variety of software domains. The wealth of published work portraying case studies in both academic and industrial settings is a sign of growing interest in this style of testing. [Gronau et al. 2000] and [Petrenko 2000] contain pointers to a wide range of MBT work.

Ibrahim K. El-Far is a doctoral student in computer sciences at the Florida Institute of Technology, Melbourne, Florida, U.S.A. (ielfar@acm.org). James A. Whittaker is an associate professor of computer sciences at the Florida Institute of Technology (jw@se.fit.edu).

We open this discussion of model-based testing with a few definitions of common terms. Some of the more popular software models are explored, and some guidelines for selecting models are presented. We then review the fundamental tasks of MBT including gathering the necessary information to build a model, the steps in building a model, and generating and evaluating tests from a model. Whenever appropriate, illustrations are given with finite state models. Subsequent sections present concise collections of important issues and considerations for model selection, construction and use. Finally, we share our views of where model-based techniques fit in the software engineering process both in today's world and in the foreseeable future.

2. Model-based Testing

2.1 Models

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.

2.2 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, statecharts, the unified modeling language (UML) and Markov chains.

2.2.1 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] 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.

The literature on formal language and automata theory is rich with information on designing, manipulating and analyzing finite state machines. This literature is a must read for software testers who pursue MBT. A classic on formal languages and automata theory is [Hopcroft & Ullman 1979]. For a different presentation and easier reading, there is [Sudkamp 1996]. A description of the two useful Mealy and Moore variations of finite state machine models can be found in these books, but the original work may also be useful [Mealy 1955], [Moore 1956]. Another original work worth reading is [Kleene 1956]. You can find a brief but insightful mention of the use of finite state machines (and other models) in requirements engineering in [Sommerville & Sawyer 1997]. You can read about these models in only a couple of testing books: [Beizer 1995] and [Jorgensen 1995]. Finally there is work on finite state machines in testing with varied tones, purposes, and audiences: [Apfelbaum & Doyle 1997], [Clarke 1998], [Fujiwara et al. 1991], [Robinson 1999 STAR], [Rosaria & Robinson 2000], [Whittaker 1997], [Liu & Richardson 2000].

Later in this paper, we will pursue finite state models in much more detail.

2.2.2 Statecharts

Statecharts 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 statecharts 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.

Statecharts are intuitively equivalent to the most powerful form of automata: the Turing machine. However, statecharts are more pragmatic while maintaining the same expressive capabilities. Statecharts are probably easier to read than finite state machines, but they are also nontrivial to work with and require some training upfront.

Work on statecharts can be found in [Harel 1987] and [Pnueli & Shalev 1991]. A sample of testing with statecharts can be read in [Thevenod-Fosse & Waeselynck 1993] and [Hong 2000].

2.2.3 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 statecharts can become components of the larger UML framework.

You can read about UML in [Booch et al. 1998]. 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], [Basanieri & Bortolino 2000], [Hartman et al. 2000], [Offutt & Abdurazik 1999], and [Skelton et al. 2000].

2.2.4 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] and [Agrawal & Whittaker 1993]. Work related to measurement and test can be found in [Whittaker & Thomason 1994] and [Whittaker & Agrawal 1994], [Walton & Poore 2000 IST], [Whittaker et al. 2000]. Other general work worth looking at include [Doerner & Gutjahr 2000] and [Kouchakdjian & Fietkiewicz 2000].

2.2.5 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.

Scarcity characterizes the grammar-based testing literature in general. Try [Duncan & Hutchinson 1981], [Maurer 1990], and [Maurer 1992] for a start.

2.2.6 Other Models

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

2.3 Finite State Machine Terminology

2.3.1 Definition

Formally a finite state machine representing a software system is defined as a quintuple *(I, S, T, F, L)*, where

- *I* is the set of inputs of the system (as opposed to input sequences).
- *S* is the set of all states of the system.
- *T* is a function that determines whether a transition occurs when an input is applied to the system in a particular state.
- *F* is the set of final states the system can end up in when it terminates.

- *L* is the state into which the software is launched.

A finite state machine can only be in one state at any one time. The occurrence of a transition from one state to another is exclusively dependent on an input in *I*.

2.3.2 Variations

There are variations of this definition. The model may be restricted to having only one starting state and one final state. Others have multiple starting and final states. In some state machines, a transition is dependent, in addition to the current state and the application of an input, on some condition external to the machine. There are extensions in which the machine can be in more than one state concurrently. Finally, the states and transitions of a hierarchical state machine can themselves represent other automata.

2.3.3 Example

Consider a simple hypothetical light switch simulator. The lights can be turned on and off using one input. The intensity of the light can be adjusted using two inputs for lowering and increasing the intensity. There are three levels of light intensity: dim, normal, and bright. If the lights are bright, increasing the intensity should not affect the intensity. The case is similar for dim light and decreasing the intensity. The simulator starts with the lights off. Finally, when the lights are turned on, the intensity is normal by default, regardless of the intensity of the light when it was last turned off.

The simulator can be in only one of four distinct states at any one time: the lights are either off, dim, normal, or bright. One way to model this is to use a finite state machine that is defined as follows:

```
“Light Switch” = (I, S, T, F, L), where:

o   I = {<turn on>, <turn off>, <increase intensity>, <decrease intensity>}
o   S = {[off], [dim], [normal], [bright]}
o   T:
        o   <turn on> changes [off] to [normal]
        o   <turn off> changes any of [dim], [normal], or [bright] to [off]
        o   <increase intensity> changes [dim] and [normal] to [normal] and
            [bright], respectively
        o   <decrease intensity> changes [bright] and [normal] to [normal]
            and [dim], respectively
        o   The inputs do not affect the state of the system under any
            condition not described above
o   F = [off]
o   L = [off]
```

Figure 1 "Light Switch" Finite State Machine Definition

2.3.4 Representation

Finite state machine models can be represented as graphs, also called state transition diagrams, with nodes representing states, arcs representing transitions, and arc-labels representing inputs causing the transitions. Usually, the starting and final states are specially marked. Automata can also be represented as matrices, called state transition matrices. There are two useful forms of state transition matrices that are illustrated for the “Light Switch” along with the corresponding state transition diagram.

	[off]	**[dim]**	**[normal]**	**[bright]**
[off]			<turn on>	
[dim]	<turn off>		<increase intensity>	

[normal]	<turn off>	<decrease intensity>		<increase intensity>
[bright]	<turn off>		<decrease intensity>	

Figure 2 "Light Switch" State Transition Matrix

	<turn on>	**<turn off>**	**<increase intensity>**	**<decrease intensity>**
[off]	[normal]			
[dim]		[off]	[normal]	
[normal]		[off]	[bright]	[dim]
[bright]		[off]		[normal]

Figure 3 "Light Switch" State Transition Matrix (another form)

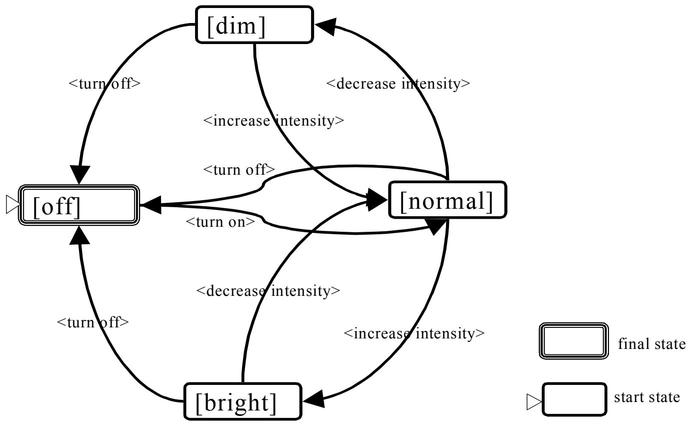


Figure 4 "Light Switch" State Transition Diagram

3. Fundamental Tasks of MBT

3.1 Understanding the System under Test

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. By applying some exploratory techniques (see [Kaner et al. 1999]) and reviewing available documents, model-based testers can gather enough information to build adequate models. The following are some guidelines to the activities that may be performed.

- *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.

- *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.

- *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.

- *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.

- *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. At first, this may be tricky since many testers and developers are not comfortable with the idea of software having users that are neither human nor programmable interfaces. This identification is a first step to study events and sequences thereof, which would add to testers' ability to diagnose unexpected test outcomes. Finally, we must single out those users whose behavior needs to be simulated according to the testing objectives.

- *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. At this point, because automation is usually intended in MBT, the tester needs to begin investigating means of simulating inputs and detecting output.

- *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.

- *Document input applicability information.* To generate useful tests, the model needs to include information about the conditions that govern whether an input can be applied by the user. For example, a button in a particular window cannot be pressed by the human user, if that window is not open and active. In addition, testers need to note changes in these conditions triggered by one of these inputs.

- *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.

- *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?

- *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.

- *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.

- *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 documentation is sufficient more often than not. Some activities like studying input applicability and input sequences may not be standard in earlier phases of the engineering process while supplying vital information to the modeling process; this is probably worth documenting. Also, documenting problems encountered during modeling and the rationale behind modeling decisions is also recommended. In the absence of any documentation, it may be worthwhile to also document all exploratory findings. Finally, it is quite reasonable to annotate the model with comments and observations especially if there is a lack of proper documentation and if the modeling tools allow it. In the end, that is what models are all about: to express an understanding of software behavior.

3.2 Choosing the Model

[Sommerville & Sawyer 1997] talk about some guidelines for choosing a model for software requirements. Some of these guidelines are summarized and adapted below for our purposes of model-based testing.

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. However, there is little or no data published that conclusively suggests that one model outstands others when more than one model is intuitively appropriate.

The following are some of the factors that can help determine what models are needed.

3.2.1 Choosing a Model

Different application domains lend themselves better to different types of models. No large-scale studies have been made to verify the claims of any particular model. Nevertheless, there are some intuitive observations that can be made. For example, to model HTML files for a web browser or mathematical expressions for a scientific calculator, a grammar is probably the quickest and easiest approach to modeling, since grammars are typically used to describe such languages. 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. Parallel systems are expected by definition to have several components that run concurrently. If the individual components can be modeled using state machines, then statecharts are a reasonable solution.

There are many more examples of this. If a system can be in one of very few states but transitions are governed by several external conditions in addition to inputs, then a model more potent than a finite state machine is needed; statecharts may be appropriate. If a system can be modeled with a finite state machine, but there is intention of statistical analysis of failure data, then Markov chains are more suitable. Likewise, if operational profiles are to guide test generation, Markov chains are good candidates (assuming the system can be modeled using state machines).

The choice of a model also depends on aspects of the system under test. For example, if we are attempting to find out if a system withstands long, complicated, non-repetitive sequences of inputs then state machines should be considered. 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. Another example of grammars as a preferred model is testing network packet processors and other protocol-based applications.

Finally, 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 is useful for this. If there is need to represent conditions under which inputs cause a particular response, but state machines are not appropriate, something like decision tables maybe more appropriate [Davis 1988].

3.2.2 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 some other form of state machine. Moreover, finite state machines are equivalent to 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 like graph theory that can help in guided testing and in setting and achieving coverage criteria when using grammars.

3.2.3 Skills, audience, and tools

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. 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 will review the model at some point, then something in between the optimal-for-testing and best-for-human-understanding is more appropriate.

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.

3.2.4 Organizational and Other Issues

As we mentioned earlier, an organization already employing models to clarify requirements and build 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.

Another 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.

There are also factors specific to environments of short development cycles. Model-based testing does 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.

3.3 Building the Model

In general, state model-based testers define high-level state abstractions and then refine these abstractions into an actual state space. Enumerating the state space by hand is formidable except when modeling some small subsystems.

State abstractions are based on inputs and information about the applicability of each input and the behavior that the input elicits. Indeed, the standard definition of a state used in various methodologies (for a good sampling of the various state-based methods see Harry Robinson's site www.model-based-testing.org) deals with these two issues: a state is defined by which inputs are applicable and which other states are reachable from that state.

The general procedure that underlies many MBT methodologies is laid out in [Whittaker 1997]:

- Make a list of the inputs (as described above in section 3.1).
- For each input document the situations in which the input can be applied by users and, conversely, the situations in which users are prevented from applying the input. As described above, these situations are referred to as *input applicability constraints*. For example, consider a simple telephone. The input "take the handset off the hook" can only be applied when the handset is hung up. (It is impossible to pick up the handset if the handset has already been picked up.) Modern graphical user interfaces create many such situations in which, e.g., modal dialog boxes force the user to respond to the dialog and disallow any other input.
- For each input document the situations in which the input causes different behaviors (or outputs) to be generated depending on the context in which the input is applied. These situations are referred to as *input behavior constraints*. For example, the input "take the handset of the hook" can cause a number of behaviors. If the context is that the phone is idle, the resulting behavior will be that the system will generate a dial tone. If the context is that the phone is ringing due to an incoming call, then the system will connect the two phones to allow conversation. Two different behaviors are caused by the same input.

Both input applicability constraints and input behavior constraints describe scenarios that need to be tested. The fact that MBT forces a tester to explicitly think these situations through thoroughly is one of its chief benefits.

Both types of constraints can be represented as sets. For example, the following sets document the constraints mentioned above.

Phone Status = {on hook, off hook}
Incoming Call = {yes, no}

Once each constraint is documented, one can form the cross product of the sets to produce the state space. Various algorithms have been offered to accomplish this so there is no need to reproduce them here. Prowell (2000) introduces a language to that manageably describes software usage models (Markov chains). Whittaker (1997) talks about building finite state models in a hierarchical manual fashion. El-Far (1999) describes a framework for automatically building finite state models from an informal specification of the system under test. There is some work that has been done on automating the generation of probabilities for Markov chains: [Walton 1995] and [Walton & Poore 2000 SPE].

The process to develop other types of model (grammars, etc.) is fairly similar despite the differences in ultimate representation. Since the goal in any case is to allow automatic generation of input sequences and verification of expected behavior, it is no great surprise that the analysis of the input domain requires specification of when inputs are allowable and what resulting behaviors can ensue.

3.4 Generating Tests

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 combinatorics. In the case of finite state machines, it is as simple as implementing an algorithm that randomly traverses the state transition diagram. The sequences of arc labels along the generated paths are, by definition, tests. For example, in the state transition diagram below, the sequence of inputs “a, b, d, e, f, i, j, k” qualifies as a test of the represented system.

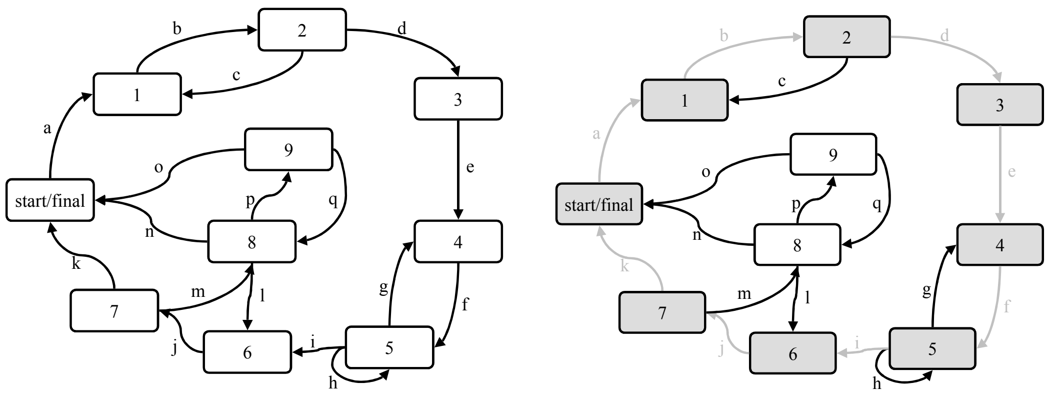


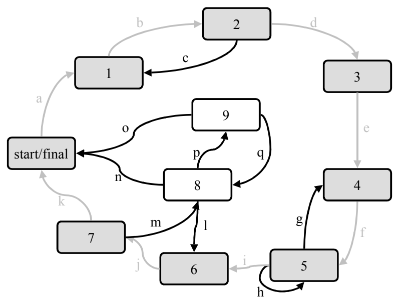
Figure 5 A state machine (left) and a test path in the state machine (right)

There are a variety of constraints on what constitutes a path to meet the criteria for tests. Examples include 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.

More useful reading can be found in [Avritzer & Weyuker 1995], [Walton & Poore 2000 IST], and [Fujiwara et al. 1991].

3.5 Running the Tests

Although, tests can be run as soon as they are created, it is typical that tests are run after a complete suite that meets certain adequacy criteria is generated. First, test scripts are written to simulate the application of inputs by their respective users (refer to figure 6 for an example). Next, the test suite can be easily translated into an executable test script. Alternatively, we can have the test generator produce the test script directly by annotating the arcs with simulation procedures calls.



```
Procedure a() {
  // Simulation code for input a
}

Procedure b() {
  // Simulation code for input a
}

Procedure c() {
  // Simulation code for input a
}
.
.
.
```

```
Procedure Test_Script() {
   // Code to drive software
   // into start state
   a();
   b();
   d();
   e();
   f();
   i();
   j();
   k();
}
```

Figure 6 Test path in state machine (left), simulation script for each input (center), and test script for illustrated path (right)

While writing the automation code, adherence to good engineering practices is required. Scripts are bound to interact with each other and evolve as the software evolves. Scripts can be used for as long as the software is being tested, so it worthwhile investing some time in writing good, efficient ones. With model-based testing, the number of simulation routines is in the order of the number of inputs, so they are generally not too time-consuming to write.

3.6 Collecting Results

Evaluating test results is perhaps the most difficult testing process. Testers must determine whether the software generated the correct output given the sequence of test inputs applied to it. In practice, this means verifying screen output, verifying the values of internally stored data and establishing that time and space requirements were met.

MBT does not ease this situation. Output and internal variables must still be checked against the specification for correctness. However, MBT does add one new dimension that is very useful in practice: the verification of state.

States are abstractions of internal data and as such can often be more easily verified. For example, the model itself will catalog each state change that occurs (or should occur) in the software as test inputs are being applied. Thus, the model acts as a very precise and detailed specification, informing the tester what inputs should be available and the values of each data abstraction that comprises the state of the software under test.

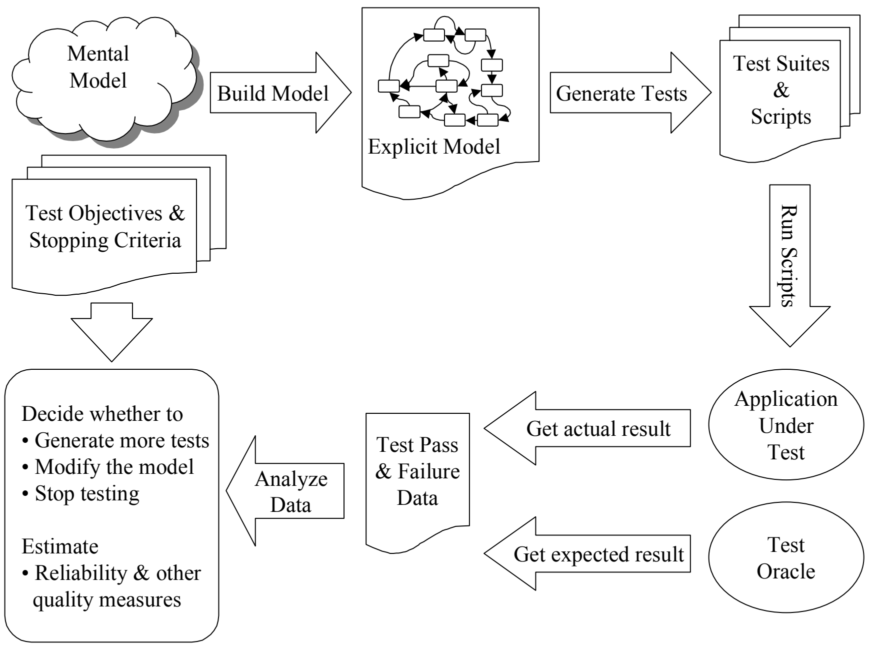


Figure 7 Some Model-based Testing Activities

3.7 Making Use of Test Results

During traditional testing, tests are conceived and executed one-at-a-time or in batch. Suppose, e.g., that a tester was able to run 400 test cases and exactly 40 bugs were found as a result. At the end of such a test, one can claim only that 400 tests were run and 40 bugs were found. There is little in the way of context to determine how the software would have fared had the 401st test case been run.

However, models not only give a good picture of what tests have been run, but also give insight into what tests have not been run. The analysis of the nature of determining the difference between these two sets forms a useful metric to determine the completeness of a test and provides stopping criteria inherent to the model [Whittaker & Thomason 1994]. For example, one may want to stop testing when there are no more non-repetitive tests described by the model.

4. Key Concerns

4.1 State Space Explosion

There is mention of state space explosion in most papers that deal with building, maintaining, reviewing, checking, and testing using models. Reading Weyuker's article on her experiences in testing with large models is a pleasant and informative exposure of state space explosion among other issues [Weyuker 1998]. Indeed this problem can never be truly appreciated without an example.

Basically, there are two ways to deal with state explosion if it cannot be faced directly, *abstraction* and *exclusion* [Whittaker 1997]. Abstraction for MBT involves the incorporation of complex data into a simple structure. For 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." Note that the abstraction captures the end result of the data that is entered so that a transition to the correct state can be made. Thus, when the data is invalid, the model represents enough knowledge to transition to the error state. However, the use of abstraction always has the effect of losing some information. In this case, we lose the information about exactly which field was entered incorrectly. We simply know that as an ensemble, the data is incorrect. Obviously, abstraction can be misused. But when used wisely, it can retain the important information and ignore superfluous or irrelevant information without loss of effectiveness.

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 inputs and exclude others. The inputs excluded by a modeler would have to be tested in some other way, obviously.

State explosion represents a serious challenge to the model-based tester but by no means does it preclude the use of finite state models, for there are no silver bullets in testing.

4.2 Meeting Coverage Criteria

[Zhu et al. 1997] is one of the better surveys of coverage criteria, but it does not adequately cover those related to MBT. There have been no comprehensive studies of the efficiency or effectiveness of the various model coverage proposals [Agrawal & Whittaker 1993], [Rosaria & Robinson 2000]. This is a gaping hole in the theory of MBT.

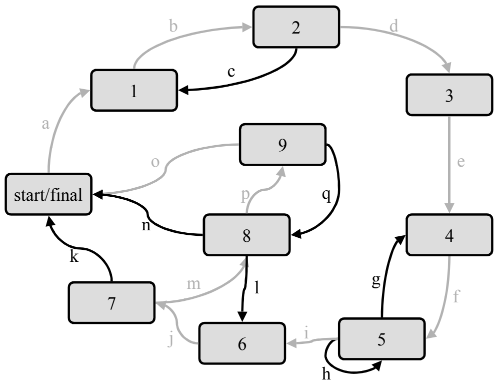


Figure 8 A state machine test path that covers all states

For state machines, coverage is expressed in terms of inputs, states, and transitions. For example, there is the criterion of covering all states (see above figure). Of course, small models will allow the practitioner the happy situation of full state and transition coverage. Larger models are problematic and until more research is performed, model-based testers must choose from the vast smorgasbord of graph coverage algorithms available [Gross and Yellen 1998].

4.3 Oracles and Automation

Automation and oracles make interesting bed fellows. On the one hand, oracles are crucial to making automation work. What good is it to execute one million tests cases if we cannot tell which ones passed and which ones failed? On the other hand, automation itself makes writing an oracle even more difficult. Because manual testing is slow, fewer tests can be performed and oracles do not have to be as comprehensive. Indeed, they must cover only those behaviors that the manual tester has time to perform. Moreover, oracles for manual testing can also be manual because there is usually time to verify screen output during the slow process of manual test execution. However, automated testing is blazing fast, making manual verification a self-defeating, non-option (that is, nevertheless, often employed in the industry due to the lack of an alternative). Further, the speed of automation means that a much larger percentage of the functionality will be covered during testing. Thus, the oracle must necessarily cover a larger portion of the possible behaviors.

There is little work that specifically addresses the oracle problem, and so it remains without so much as guidelines for a solution. For now, there can only be partial oracles in practice. Examples of an oracle include competing products, different versions of the same product in which the feature under test did not experience significant change, and different implementations developed by the same vendor. In addition, there are practical tricks such as replacing correctness with plausibility: if the test outcome appears to be within a reasonable range from the correct result, then the test passes. For further reading, try [Parnas 1998] and [Weyuker 1998].

5. Closing Thoughts

5.1 Attractive Attributes 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. For a sample of such studies, consider the works Rosaria and Robinson (2000) on testing graphical user interfaces; Agrawal and Whittaker (1993) on testing embedded controller software; and that of Avritzer and Larson (1993) and Dalal et al. (1998) on testing phone systems.

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. For example, confusion as to whether the system under test needs to satisfy a particular requirement can be resolved by examining the model. This is due in part to the model being an artifact that is readable by both developers and non-technical customers and their advocates. 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.

5.2 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.

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.

5.3 MBT in Software Engineering: Today and Tomorrow

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. However, software testers of today have a difficult time planning such a modeling effort. They are victims of the ad hoc nature of the development process where requirements change drastically and the rule of the day is constant ship mode. That is why we have seen the application of explicit model-based testing limited to domains in which ship times are less hurried and engineering rigor is more highly valued. Modeling is a nontrivial matter and testers who have only a few hours or days to test will most often opt to maintain their models in their head and perform testing manually.

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

But 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.

Acknowledgements

The authors would like to thank Cem Kaner of the Florida Institute of Technology for some suggestions and comments on an early draft. Alan Jorgensen of the Florida Institute of Technology shared some ideas on finite state model building. Harry Robinson of Microsoft has a site on model-based testing that constituted a valuable resource (http://www.model-based-testing.org). David Gelperin of Software Quality Engineering was gracious enough to share some of his literature pointers on model-based testing. Sam Guckenheimer of Rational, through a special-interest mailing list, shared some pointers to UML-related testing work. Finally, Alvin Flanegin of Intel reviewed and criticized the manuscript in its near-final form. Substantial insight was gained from the anonymous reviewers.

Ibrahim K. El-Far has been partially supported by an IBM Center for Advanced Studies Fellowship while preparing this manuscript. James A. Whittaker's work is partially supported by grants from Microsoft Research and Rational Software Corporation.

References

[Abdurazik & Offutt 2000]	Aynur Abdurazik and 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, October 2000.
[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.
[Apfelbaum & Doyle 1997]	Larry Apfelbaum and J. Doyle. Model-based testing. *Proceedings of the 10th International Software Quality Week (QW 97)*, May 1997.

[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.
[Avritzer & Weyuker 1995]	Alberto Avritzer and Elaine J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9): 705-715, September 1995.
[Basanieri & Bortolino 2000]	F. Basanieri and A. Bertolino. A practical approach to UML-based derivation of integration tests. *Proceedings of the 4th International Software Quality Week Europe (QWE 2000)*, Brussels, Belgium, November 2000.
[Beizer 1995]	Boris Beizer. Black-Box Testing: *Techniques for Functional Testing of Software and Systems*. Wiley, May 1995.
[Binder 2000]	Robert V. Binder. *Testing object-oriented systems*. Addison-Wesley, Reading, MA, USA, 2000.
[Booch et al. 1998]	Grady Booch, James Rumbaugh, and Ivar Jacobson. *The unified modeling language*. Documentation Set, Version 1.3, Rational Software, Cupertino, CA, USA, 1998.
[Chow 1978]	Tsun S. Chow. Testing design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3): 178-187, May 1978.
[Clarke 1998]	James M. Clarke. Automated test generation from a behavioral model. *Proceedings of the 11th International Software Quality Week (QW 98)*, May 1998.
[Dalal et al. 1998]	S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott. Model-based testing of a highly programmable system. *Proceedings of the 1998 International Symposium on Software Reliability Engineering (ISSRE 98)*, pp. 174-178, Computer Society Press, November 1998.
[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.
[Doerner & Gutjahr 2000]	K. Doerner and W. J. Gutjahr. Representation and optimization of software usage models with non-Markovian state transitions. *Information and Software Technology*, 42(12):815-824, September 2000.
[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 1999]	Ibrahim K. El-Far. *Automated Construction of Software Behavior Models*. Master's Thesis, Florida Institute of Technology, May 1999.
[Fujiwara et al. 1991]	Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6): 591-603, June 1991.
[Gronau et al. 2000]	Ilan Gronau, Alan Hartman, Andrei Kirshin, Kenneth Nagin, and 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.
[Harel 1987]	D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3): 231-274, 1987.
[Hartman et al. 2000]	J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, August 2000.
[Hong 2000]	Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo Hwan Bae and Hasan Ural. A Test Sequence Selection Method for Statecharts.*The Journal of Software Testing, Verification & Reliability,* 10(4): 203-227, December 2000.

[Hopcroft & Ullman 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[Jorgensen 2000] Alan Jorgensen and James A. Whittaker. An API Testing Method. *Proceedings of the International Conference on Software Testing Analysis & Review (STAREAST 2000)*, Software Quality Engineering, Orlando, May 2000.

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

[Kaner et al. 1999] Cem Kaner, Hung Quoc Nguyen, and Jack Falk. *Testing computer software*. 2nd ed., Wiley, April 1999.

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

[Kleene 1956] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pp. 3-42, Princeton University Press, Princeton, NJ, USA, 1956.

[Kouchakdjian & Fietkiewicz 2000] A. Kouchakdjian and R. Fietkiewicz. Improving a product with usage-based testing. *Information and Software Technology*, 42(12):809-814, September 2000.

[Liu & Richardson 2000] Chang Liu and Debra J. Richardson. Using application states in software testing (poster session). *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, p. 776, ACM, Cambridge, MA, USA, 2000.

[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.

[Mealy 1955] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5): 1045-1079, 1955.

[Moore 1956] E. F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, pp. 129-153, Princeton University Press, NJ, USA.

[Offutt & Abdurazik 1999] Jeff Offutt and Anyur Abdurazik. Generating test cases from UML specifications. *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML 99)*, Fort Collins, CO, USA, October 1999.

[Paradkar 2000] Amit Paradkar. SALT: an integrated environment to automate generation of function tests for APIs. *Proceedings of the 2000 International Symposium on Software Reliability Engineering (ISSRE 2000)*, October 2000.

[Parnas 1998] Dennis K. Peters and David L. Parnas. Using test oracles generated from program documentation. IEEE Transactions on Software Engineering, 24(3): 161-173, March 1998.

[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*).

[Pnueli & Shalev 1991] A. Pnueli and M. Shalev. What is in a step: on the semantics of statecharts. *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, Japan, September 1991.

[Poore et al. 1993] J.H. Poore, H.D. Mills, and D. Mutchler. Planning and certifying software reliability. *IEEE Software*, 10(1): 88-99, January 1993.

[Poore et al. 2000] J. H. Poore, G. H. Walton, and James A. Whittaker. A constraint-based approach to the representation of software usage models. *Information and Software Technology*, 42(12):825-833, September 2000.

[Prowell 2000] S. J. Prowell. TML: a description language for Markov chain usage models. *Information and Software Technology*, 42(12): 825-833, September 2000.

[Quatrani 1998] T. Quatrani. *Visual Modeling with Rational Rose and UML.* Addison Wesley Longman, 1998.

[Rational 1998] Rational Software Corporation. *Rational Rose 98i: Using Rose*. Rational Rose Documentation, Revision 6.0, Rational, December 1998.

[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.

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

[Skelton et al. 2000] G. Skelton, A. Steenkamp, and C. Burge. Utilizing UML diagrams for integration testing of object-oriented software. *Software Quality Professional*, June 2000.

[Sudkamp 1996] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Second edition. Addison Wesley Longman, November 1996.

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

[Thevenod-Fosse & Waeselynck 1991] Pascale Thevenod-Fosse and Helene Waeselynck. An investigation of statistical software techniques. Journal of Software Testing, Verification & Reliability, 1(2): 5-25, July/September 1991.

[Thevenod-Fosse & Waeselynck 1993] Pascale Thevenod-Fosse and Helene Waeselynck. STATEMATE Applied to Statistical Software Testing. *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA 1993)*, pp. 99-109, ACM, Cambridge, MA, USA, 1993.

[Walton 1995] Gwendolyn H. Walton. *Generating Transition Probabilities for Markov Chain Usage Models*, Ph.D. Thesis, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, USA, 1995.

[Walton et al. 1995] Gwendolyn H. Walton, Jesse H. Poore, and Carmen J. Trammell. Statistical testing of software based on a usage based model. *Software: Practice and Experience*, 25(1): 97-108, January 1998.

[Walton & Poore 2000 SPE] Gwendolyn H. Walton and Jesse H. Poore. Generating transition probabilities to support model-based software testing. *Software: Practice and Experience*, 30(10):1095-1106, August 2000.

[Walton & Poore 2000 IST] G. H. Walton and J. H. Poore. Measuring complexity and coverage of software specifications. *Information and Software Technology*, 42(12):815-824, September 2000.

[Weyuker 1982] Elaine J. Weyuker. On testing non-testable programs, The Computer Journal, 25(4), 1982.

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

[Whittaker 1992] James A. Whittaker. *Markov chain techniques for software testing and reliability analysis*. Ph.D. dissertation, University of Tennessee, Knoxville, May 1992.

[Whittaker 1997] James A. Whittaker. Stochastic software testing. *Annals of Software Engineering*. 4:115-131, August 1997.

[Whittaker 2000] James A. Whittaker. What is software testing? And why is it so difficult. *IEEE Software*, 17(1):70-79, January/February 2000

[Whittaker & Agrawal 1994] James A. Whittaker and K. Agrawal. The application of cleanroom software engineering to the development of embedded control systems software. *Proceedings of the 1994 Quality Week*, May 1994.

[Whittaker et al. 2000] James A. Whittaker, Kamel Rekab, and Michael G. Thomason. A Markov chain model for predicting the reliability of multi-build software. *Information and Software Technology*, 42(12): 889-894, September 2000.

[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.
[Zhu et al. 1997]	Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366-427, 1997.