Hybrid Model Based Testing for Mobile Applications

Haeng-Kon Kim
School of Information Technology, Catholic University of Daegu, Korea
hangkon@cu.ac.kr

Abstract

Mobile application 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 is a new and evolving technique for generating a suite of test cases from requirements. Testers using this approach concentrate on a data model and generation infrastructure instead of hand-crafting individual tests.

In this paper we discuss the characteristics of automotive model-based development processes, the consequences for test development and the need to reconsider testing procedures in practice. Furthermore, we introduce the test tool HMBT (Hybrid Model Based Testing) which masters the complexity of model-based testing in the automotive domain. To illustrate this statement we present a small mobile applications case study. HMBT is based on graphical test models that are not only easy to understand but also powerful enough to express very complex, fully automated closed loop tests in real-time.

Keywords: Hybrid Model Based Testing, Mobile Applications Testing, Verification and Validation, Quality Assurance, Testing Strategy

1. Introduction

Model-based testing refers to deriving a suite of test cases from a model that represents the behavior of a software system. By executing a set of model-based test cases, the conformance of the target system to its specification can be validated. One commonly used modeling diagram for that purpose is state machine diagram. According to UML (Unified Modeling Language) specification [1], a state machine diagram consists of a set of states and transitions. A transition is triggered when an event occurs and a condition associated with the transition is satisfied.

When a transition is triggered, the associated actions will be performed which may lead to a state change of the system.

These model-based technologies allow the development of high-level models that can be used for simulation in very early stages of the development process. This in turn is important since automotive development is an interdisciplinary business with software, electrical, and mechanical engineering aspects inextricably entwined. Graphical models and simulation of such models allows engineers to find a common functional understanding early in the design phase. So, model-based development improves communication within development teams, with customers, or between car manufacturers and suppliers. It reduces time to market through component re-use and reduces costs by validating systems and software designs up
front prior to implementation. Consequently specification, since the models illustrate the required functionality in an executable manner. Model-based development provides a development process from requirements to code, ensuring that the implemented systems are complete and behave as expected. Model based development allows segregation of concerns; technical aspects such as fixed-point scaling (i.e., the transformation of floating-point algorithms to fixed point computations), calibration data management, and the representation of signals in memory are separated from the core algorithms thereby keeping the models as lean as possible.

These features are useful to the designers for modeling the dynamic behavior of event-driven systems such as communication protocols or graphical user interface systems [2]. Each feasible path of transitions [3] within a state machine diagram is considered as an operational scenario of the system under test. Thus, the instances of the operational scenarios will form a suite of test cases for model-based testing. However, since cycles within the state machine diagram will lead to infinite number of feasible paths of transitions, exhaustive testing is usually impossible. Therefore, an important issue is to decide which feasible paths should be selected for testing. A default criterion of adequate testing with a state machine diagram is that all transitions in the diagram are covered by the test executions. This is called all-transitions coverage criterion [4, 5] which means each transition specified in the state machine diagram is triggered at last once by executing the test cases.

In this paper we discuss the characteristics of automotive model-based development processes, the consequences for test development and the need to reconsider testing procedures in practice. Furthermore, we introduce the test tool HMBT (Hybrid Model Based Testing) which masters the complexity of model-based testing in the automotive domain. To illustrate this statement we present a small mobile applications case study. HMBT is based on graphical test models that are not only easy to understand but also powerful enough to express very complex, fully automated closed loop tests in real-time.

We are proposing to apply formal concept analysis (FCA) [9] to analyze the association of a set of feasible paths with a set of transitions specified in the state machine diagram, and to organize them to form a concept lattice. The concept lattice structure is used for analyzing the transition coverage of the feasible paths. With the concept analysis mechanism, our approach is able to reduce the test suite whilst satisfying the all-transitions coverage criterion for model-based testing.

2. Related Works

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. Classification of Model Quality Goals

When Model-Based Methodologies started to gain steam, it is only natural that good practices were formed to assure modeling is being done correctly. The 6C approach focuses on the quality of the model itself and how well it can be used with tools, languages, environments, and users [5].

C1: Correctness. Does the model include the right elements and correct relations between them, including correct statements about the domain? Does the model violate any rules or conventions (syntax)?

C2: Completeness. Does the model have all of the necessary information and is it detailed enough according to the purpose of the model?

C3: Consistency. Are there consistencies and no contradictions in the model?

C4: Comprehensibility. Is the model understandable by the intended users; humans and tools alike?

C5: Confinement. Does the model agree with all corresponding diagrams and is the model at the right abstraction level?

C6: Changeability. Does the model support changes or improvements so the model can be evolved rapidly or continuously?

![6C Approach](image)

**Figure 1. 6C Approach [5]**

2.3. Model-based testing in practice

Automated testing of software usually means writing small programs that attempt to check some aspect of a system. This is error-prone and turns to drudgery for a large system. But suppose that you could instead encode the same aspects in a way that your test programs could be automatically generated and run – that’s model-based testing. What is the essential difference between MBT and other kinds of testing? In other approaches, even if using an automated test harness, the test engineer first conceptualizes each test case. The next step is to document it for manual input, or to write and then run a test driver program. In MBT, the test engineer develops a model, which supports automated production and evaluation of test cases.
A program that embodies a testing strategy produces test cases from the model, which are usually run and evaluated in an automated test harness.

For example, suppose we’re developing a sports watch that simultaneously records time, distance, and heart rate. Under a traditional testing approach, we’d probably prepare a test plan organized around features and usage modes, and then choose specific scenarios and user actions as test cases. We’d probably have both lab tests (the watch would be embedded in a test fixture to ease control and observation) and field tests, where the tester would wear the watch and note its response in defined test cases. With MBT, the test engineer produces a test model that specifies the required response to user inputs under all operational modes. Then, a test generation program would identify each event sequence that starts and ends the same state, and determine the necessary user inputs and expected result at each step. The particular test case generation program determines the specifics of how the model is represented. For example, some systems use an event/state table or similar specification language; others accept source code with certain tags. There are many variations, and MBT is typically used in concert with traditional test approaches. Some tools have the ability to generate expected results along with the test inputs—a capability critical for industrial use.

Software reliability engineering (SRE) has been extensively studied and applied to good effect. SRE requires test suites where the relative frequency of operations in a test suite is a statistically accurate representation of actual field use. However, most applications of SRE simply tweak traditional testing strategies by requiring that the number of tests of a certain feature is in proportion to the expected field use of that feature.

To support SRE with MBT, the test model must represent the operations of interest and their expected relative frequency. If there are sequential constraints on input sequence, the model must also support a Markov chain or something like it to represent the relative frequency of each allowed operation sequence.

Along with the growing functionality and the introduction of model-based development processes, the demands on quality assurance have also increased. In terms of testing, model-based development enables system engineers to test the system in a virtual environment when they are inexpensive to fix, i.e., before the code is actually implemented or integrated on the final hardware (which is called ECU or electronic control unit). However, in practice there are just a few testing procedures that address the automotive domain-specific requirements of model-based testing sufficiently. The applied test methods and tools are often proprietary, ineffective and require significant effort and money. There was no need for dedicated functional testing methods because the functional complexity was comparatively low. With the increasing popularity of model-based development the engineering discipline of automotive model-based testing has been neglected for a long time. With the promise of model-based development to prove concepts at early development stages by means of executable models it was assumed that testing those models and the derived code is less important and therefore would not require new.

Model-based testing is a system testing technique [2, 3] that derives a suite of test cases from a model representing the behavior of a software system. Being the most formalized component of UML, state machine diagrams have been used as a basis for generating test data [4, 5, 6]. Therefore, state machine diagrams can be readily used by system domain experts to express and analyze behavioral requirements and thus provides the software developer with a means for early validation of requirements [7].

A number of researchers have proposed coverage criteria for test data selection from UML state machine diagrams. Some of the well-established criteria [8] include all-states coverage, full predicate coverage, all-transitions coverage, all-transition-pairs coverage, and complete sequence coverage. Since the all-transitions coverage criterion means every transition is...
covered at least once, it implies the satisfaction of both all-states coverage and full predicate coverage criteria. The all-transition-pairs coverage criterion in general produces $O(n^2)$ test cases [9] where $n$ is the number of transitions. Thus for large state machine models, this criterion may not be practical because it requires too many test cases. The complete sequence coverage relies on the domain knowledge of the requirements engineer in choosing the testing sequences. There may exist some cases that these testing sequences are redundant or cannot fully exercises all the transitions in a state machine diagram. Therefore, among all these coverage criteria, we choose the all-transitions coverage as the coverage adequacy criterion for selecting the test cases.

3. Hybrid Model Based Testing for Mobile Applications

3.1. Understanding the Mobile Applications System under Test

The requirement common to most styles of testing is a well-developed understanding of what the software accomplishes, and HMBT 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 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.

- **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.

3.2. Designing Tests

Subsequent to construction of a model - either complete or partial - the issue of test generation can be addressed. The test objective is to verify that the system will behave properly when a sequence of user actions occurs. For the purposes of this paper, a test script is defined as the entire sequence of actions required to create a complete user scenario for the system - from start-up through all of the actions and ending with shut-down. A test script can be decomposed into a series of individual test primitives that accomplish specific actions. The primitives will be combined in a specific sequence to provide a test script that verifies a unique use scenario for the product. Test primitives will typically fall into one of the following categories:

1) **Provide** a stimulus to the system. This is the most obvious - it controls movement from one state to another in the behavioral models. It can be a user action like selecting a button on a GUI, invoking a function of an API, or dialing a number on a telephone. The action can be directly controlled from the test execution environment.

2) **Verify** that the system responded correctly. Verification can be difficult to accomplish because the system’s response must be determined and then compared to an expected response and/or verify that we are in the correct ‘state’ in our system. The degree that verification is used and the means for performing verification will vary widely with application type and test objectives. Some examples of verification are: comparing the text in a window on a GUI, checking that a dialed phone rings, comparing the return value of a function, or establishing that a billing record is generated after a call. Not all actions require direct verification - sometimes the fact that the next stimulus will be accepted is an acceptable means of verifying that the system is currently in the correct state.

3) **Set-up** the system testing environment. Tests will often need to control the environment so that the next action will follow a predictable path. In many environments there are situations that have several potential outcomes from the same input. These are often due to multiple resources being available for the action. To make this situation deterministic, and therefore easier to test, the environment can be temporarily pre-allocated or constrained in order to force a specific sequence or response to occur. Examples of this include: making a phone line busy so that the call forwarding feature on it can be tested and changing the status on an accounting record so that an error condition can be tested.

4) **Report and/or log** the results. Depending on the degree of automation in the test execution environment commands can be embedded that will report the test results to a reporting system. These embedded commands range from simple print statements for log files, all the way to sophisticated inter-process communications with a test management system. In any situation, the means to record and analyze results should be planned. There are several approaches that can be used to develop tests from a model [7, 8] Central to most of these is the concept of a path. A path is a sequence of events or actions that traverse through the model defining an actual use scenario of the system. Each element in a path, a transition or state, can have some test primitives associated with it. The primitives will define what test actions are required to move the system from its current state to the next state, verify that the state is reached or check that the system has responded properly to previous inputs. Once a path through the
model has been defined, a test script can be created for that path. When this script is applied to the actual system, the actual system will follow the same sequence (or path) as defined by the model path from which the test script was extracted. This process can then be repeated for another path, which defines another use scenario, and verifies another sequence of actions. Many methods can be used to select paths, each with its own distinct objectives and advantages. Operational profiles, reliability/criticality data, and switch coverage all provide different tradeoffs to the type of tests and the resulting coverage. Other criteria include the quality or thoroughness desired in the tests. The test objective might range from the need for a set of tests to verify basic functionality all the way to a complete product verification suite.

3.3. Hybrid Model Based Testing Process

A test suite is a finite set of test cases. A test case is a finite structure of input and expected output: a pair of input and output in the case of deterministic transformative systems, a sequence of input and output in the case of deterministic reactive systems, and a tree or a graph in the case of non-deterministic reactive systems. The input part of a test case is called test input. In general, test cases will also include additional information such as descriptions of execution conditions or applicable configurations, but we ignore these issues here. A generic process of model-based testing then proceeds as follows (Figure 2).

![Figure 2. Hybrid Model Based Testing Process](image)

**Step 1.** A model of the SUT is built on the grounds of requirements or existing specification documents. This model encodes the intended behaviour, and it can reside at various levels of abstraction. The most abstract variant maps each possible input to the output “no exception” or “no crash”. It can also be abstract in that it neglects certain functionality, or disregards certain quality-of-service attributes such as timing or security.

**Step 2.** Test selection criteria are defined. In general, it is difficult to define a “good test case” a-priori. Arguably, a good test case is one that is likely to detect severe and likely failures at an acceptable cost, and that is helpful with identifying the underlying fault. Unfortunately, this definition is not constructive. Test selection criteria try to approximate this notion by choosing a subset of behaviours of the model. A test selection criterion possibly informally describes a test suite. In general, test selection criteria can relate to a given functionality of the system (requirements based test selection criteria), to the structure of the model (state coverage, transition coverage, def-use coverage), to stochastic characterizations such as pure randomness or user profiles, and they can also relate to a well-defined set of faults.
Step 3. Test selection criteria are then transformed into test case specifications. Test case specifications formalise the notion of test selection criteria and render them operational: given a model and a test case specification, some automatic test case generator must be capable of deriving a test suite (see step 4). For instance, “state coverage” would translate into statements of the form “reach _” for all states _ of the (finite) state space, plus possibly further constraints on the length and number of the test cases. Each of these statements is one test case specification.

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

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

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

Step 5-2. A verdict is the result of the comparison of the output of the SUT with the expected output as provided by the test case. To this end, the output of the SUT must have been abstracted. Consider the example of testing a chip card that can compute digital signatures [3].

3.2. Modeling of Hybrid Model-Based Testing (HMBT)

Hybrid Model-Based Testing (HMBT) can be defined as a methodology that uses models of systems under test (SUT) in test authoring. Test authoring usually takes the form of automatic test generation from models but can also happen as a result of SUT model simulation, where a user defines the stimuli sent to the system. The following picture describes one of the more common flavors of the Hybrid Model-Based Testing as in Figure 3.
The process is specification-based. A user creates a model of the system under test using some formal modeling language supported by the tool based on the specification. This model is used to generate test cases automatically. The quality and size of the generated test suite depends on the directives provided by the user (e.g., coverage criteria or model exploration depth). Each test case contains both the stimuli to be sent to the system under test and the prediction of the outcome of applying those stimuli. The test execution prediction is based on the logic described by the model. The test cases may need to be translated to the executable scripts that are used by the test execution engine of choice. In case test execution results do not match the predicted one - there is a bug that needs to be fixed. There are many various flavors of MBT. Some approaches do not generate a test suite in advance but create the next step of the test based on the execution result of the previous step. Other approaches do not predict the outcome of test cases which simplifies the modeling task significantly as in Figure 4.

4. Applying HMBT Test Suite to Mobile Application

In this section, we use a simplified behavior model of an MP3 player to illustrate the mechanism of reducing model-based test suite with HMBT. With reference to the all-transitions coverage criterion, the following questions are to be addressed in this section:

- **Sufficiency of test coverage**: Is every transition specified in the state machine diagram triggered at last once by the selected test cases?
- **Reduction of test suite**: How could we keep the test suite minimal whilst maintaining sufficient test coverage?

Figure 5 depicts a state machine diagram of a simplified MP3 player. The model comprises of four states: Off, Ready, Playing, and Paused. It describes the set of events that will trigger the transitions for changing the state of the MP3 player. For the ease of explanation, we labeled each transition with an identifier.
By traversing the state machine diagram, a set of feasible paths of transitions can be obtained. However, because of the iterative nature of the given state machine diagram, there are infinite number of feasible paths. Suppose that we only consider those feasible paths which have depth ≤ 5 and at most one transition can be traversed twice by the same feasible path. Figure 6 shows the resultant set of 13 feasible paths in form a transition tree [3].

4.2. Sufficiency of Transition Coverage

In the context of transition coverage, the feasible paths can provide sufficient coverage if when test cases are executed according to the sequence specified in the feasible paths, each transition specified in the state machine diagram will be triggered.
at least once. With reference to a concept lattice structure, a simple indication for sufficiency of transition coverage is:

\[ \text{AttributeLabels}(\text{Bottom}) = \emptyset \wedge \]

\[ \text{ObjectLabels}(\text{Bottom}) = \emptyset. \]

That implies every transition \( t \) is covered by some feasible path \( p \). Therefore, as shown in Figure 5, the feasible paths can provide sufficient coverage of all the transitions.

### 4.3. Reduction of Test Suite

A set of feasible paths is considered to be minimal if any of the feasible paths is removed, some transitions in the given state machine diagram are not covered by the remaining feasible paths. With the notion of concept lattice, our approach can determine a minimal set of feasible paths via the following steps.

**Step 1: Identification of the significant feasible paths**

A feasible path is considered to be significant if it can trigger some transitions which are not covered by other feasible paths.

With reference to a concept lattice structure, a feasible path \( p \) is significant if:

1. \( p \in \text{ObjectLabels}(c) \); and
2. there is no other concept \( c' \) such that \( c \geq c' \geq \text{Bottom} \)

In Figure 7, the concept nodes associated with significant feasible paths (i.e., those nodes that are closest to the Bottom concept) are highlighted on the concept lattice. This set of significant feasible paths \{P03, P05, P07, P08, P09, P10, P12\} is sufficient enough to cover all the transitions specified in the state machine diagram of the MP3 player. Therefore, those non-significant feasible paths \{P01, P02, P04, P06, P11, P13\} can be discarded, and the resulting concept lattice is restructured as shown in Figure 6.

![Figure 7. Revised concept lattice with P01, P02, P04, P06, P11, P13 removed](image-url)
Step 2: Identification of the redundant feasible paths

The set of feasible paths can further be reduced by excluding those redundant feasible paths, if they exist.

A feasible path $p$ is considered to be redundant if:

(i) $p$ is significant; and

(ii) $p \in \text{ObjectLabels}(c)$, where

(iii) $\text{AttributeLabels}(c) = \emptyset$

That means, there is no transition that is solely covered by that feasible path $p$. All the transitions covered by $p$ can also be covered by other feasible paths.

In case there is more than one redundant feasible path, the one with least number of intent elements (i.e., covering least number of transitions) will be selected for removal first. This step is repeated until there is no more redundant feasible path.

As revealed from the concept lattice in Figure 8, there are four potential redundant paths: P03, P05, P09, and P10. We select P03 for removal first and the concept lattice is revised as shown in Figure 7.

![Figure 8. Revised concept lattice with P03 removed](image)

Then, after we further select P05 for removal, there is no more redundant feasible path in the resultant concept lattice as shown in Figure 8. Therefore the remaining set of feasible paths \{P07, P08, P09, P10, P12\} is considered to be minimal whilst fulfilling the all-transitions coverage criterion. With that result, we can further develop a specification of test cases (listed in Figure 9) corresponding to the minimal set of feasible paths.
Figure 9. Revised concept lattice for the minimal set of feasible paths

4. Evaluation

4.1. Outlook for MBT

Although its technology has some inherent differences, the digital logic/device industry provides a broad roadmap for MBT. Sophisticated, model-driven, and automated verification has been a mainstay of chip development over all its vertical and horizontal segments. Integrated logic design and verification tools are commonly used. The cost for sustained use of MBT is often perceived to be excessive relative to traditional testing. Although its economic payback has been proven many times, to be effective, test engineering with MBT has to be on equal footing with systems engineering and take place upstream of the actual testing effort. Also, the recent “agile” emphasis on short-term and immediate results is often seen to be in conflict with the once-removed artifacts of MBT. This “cultural” friction can result in the exclusion of MBT or its abandonment after an initial experiment. Also, for firms not willing to develop their own MBT systems, gaps in the tool marketplace can be a deterrent.

Figure 10. Implementation of MBT
Recent interest in model-driven architecture (MDA) has renewed interest in MBT, as MBT provides another way to leverage MDA models and tools, and requires a similar design-first discipline. MDA repositories and program-generation technologies can also support MBT, but time will have to tell how much real synergy will occur.

If you use or are looking to use SRE to support testing and release, MBT can be very useful. Once in place, a suitable MBT system can generate a statistically realistic yet unique test suite in minutes. In contrast, it isn’t unusual to take weeks or months to manually produce a statistically realistic and unique test suite. In practice, this means manual reliability test suites are produced once and re-run, repeating exactly the same experiment over and over. With MBT, the input space can be re-sampled for each test run resulting in higher coverage, more effective testing, and greater confidence in reliability estimates.

4.2. 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. Conclusion and Future Works

One trend that is present today and will surely be present in the future is the improvement of the automatic code generation. It is critical to be able to generate good, reusable code that can potentially be used in a software verification process. Another trend is the use of UML to predict performance metrics and dynamically change the model if needed to reach performance goals. Unfortunately the auto-generated code is sometimes unusable so a goal in the future is to increase the quality of the code.
Model-Based Software Development has many advantages over its few disadvantages. The idea of automatic code generation and effective communication between shareholders and stakeholders is something that is continually being researched. With the right process, Model-Based Development can conquer high complexity systems with more modeling and less coding ultimately being an effective way to develop software.

Good mobile applications 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.

HMBT 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.

HMBT, 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

This work (Grants No.C0024433) was supported by Business for Cooperative R&D between Industry, Academy, and Research Institute funded Korea Small and Medium Business Administration in 2012.

References


