Experiences in Testing Pocket PC Applications

Ibrahim K. El-Far, Herbert H. Thompson, and Florence E. Mottay
Florida Institute of Technology

Keywords

Pocket PC; model-based testing; finite state models; case study

1. Introduction

1.1 Remarks on Testing with Software Models

Model-based testing (or MBT) techniques are deeply rooted in such fields as phone switches and computer hardware components. However, their value remains vague in the software industry, despite their apparent intuitive appeal. Perhaps this can be attributed to poor understanding of the underlying principles and concepts of testing with models. Conceivably, it can also be attributed to a troublesome paradigm shift from what is widely practiced today. On the other hand, the fact remains that there is an obvious shortage of useful or insightful case studies. Further, there is barely any work that faithfully details the goals, activities, and risks involved to the average test professional who is expected to work with these methods. Indeed, many of these professionals today are oblivious to the very existence of MBT, and, those who are aware of it are, at best, highly doubtful of its value and the kind of returns on investment it presents. Recently, there has been a rise in the number of researchers and testers willing to take the time to investigate the models and methods of the paradigm (El-Far 2001).

These investigations seem to have started to pay off. Over the past few years, there have been many success stories about employing models to steer various testing activities such as test generation and test result evaluation (El-Far and Whittaker 2001). Such reports have generated a lot of enthusiasm with the popularization of object-oriented technologies and the advent of model-based design and specification methods and tools. As a result, we have been witnessing a rapid growth of the relevant body of literature since the 1990s. As with any developing field, the literature is affected by the lack of a common body of knowledge and a standard set of terms that are precisely defined or that everyone uses consistently. However, there are numerous lessons to be learned and many observations to be made on the model-based testing paradigm as a whole, notwithstanding the differences among various types of models.

For example, the literature has some pointers as to the benefits of model-based testing, many of which seem to agree with intuition (Robinson, 2000). For instance, the underlying model is a formal, precise expression of a tester's understanding of how the software is supposed to work. When such an understanding is written out to a structure that others can review, update, modify, and influence with their own understanding of the software under test, many problems can be solved. The model becomes a point of reference for the testing team, an aid to presenting results to non-technical staff, and a form of documentation that reflects the most recent build of the system – a living specification. Another benefit that is typical of several models is that they have a substantial and rich theoretical background that makes numerous tasks such as generating large suites of tests fairly easy to automate. Examples of this are the theories of graphs (Gross & Yellen, 1998) and automata (Ullman & Hopcroft, 1979) for finite state machines and stochastic theory for Markov-chain models (Kemeny & Snell, 1976).

It is unfortunate that, due to many reasons, that we rarely see reports of failure or articles that contain warnings of pitfalls and tips on what to expect. For this reason, we are usually left to deduce most drawbacks from reports and from hard experience. Perhaps the thorniest such issue is one that plagues all forms of automated testing, namely, the oracle problem: how do we build an automated mechanism that checks the outcome of tests against the required behavior? The absence of an oracle is an obstacle to the automated execution of long tests or large suites of tests, both acclaimed by the field as major benefits of the approach (El-Far, 2001). Another significant drawback is the substantial investments, time and personnel, that typically have to go into building, reviewing, and maintaining models. Even with the smallest models, precious time will be lost before testers start to reap any fruit. Consequently, short development cycles, major delays in development, postponing testing activities until after components are developed can all potentially reduce the value of using models.

An interesting observation that can be drawn from the literature is that success reports seem to always come from only a few application domains: phone switch software (Avritzer & Larson, 1993), embedded software such as that in hardware controllers (Agrawal & Whittaker, 1993), and graphical user interfaces (Rosaria & Robinson, 2000), to mention some of the more typical domains. This is very encouraging for those who are considering employing models in testing these and other similar systems, although they would have to keep in mind that the results at our disposal are certainly not beyond doubt. So, not too surprisingly, when we were about to embark on a testing endeavor of some Pocket PC applications, we were enthusiastic and encouraged by what we know from the works of others and our earlier finite-state model based testing experiences. We shall elaborate on this later, but, first, we will briefly introduce the project in concern.

1.2 The Project at a Glance

Pocket PC is a Microsoft platform for handheld devices such as palmtops (Microsoft Inc. Official Website). It is powered by Windows technologies and has the look and feel of a scaled down version of a member of the Windows family of operating systems. Pocket PC devices ship with a collection of built in utilities. These are small applications that are design to be familiar to the Windows desktop user. They include, for example, Pocket Word (a simplified word processor) and Pocket Outlook (email manager and organization utility). Other applications can be added by the user or by a third party vendor.

Several months before the planned release date, when the product had reached a reasonable degree of stability, Microsoft contracted our group at the Center for Software Engineering Research in the Florida Institute of Technology to test five standalone components packaged with Pocket PC: Contacts, Calendar, Inbox, Connectivity Manager, and Pocket Word, all of which we will describe in some detail in a later section. They were particularly interested in seeing us apply finite-state model based techniques that were developed in part by researchers at the Center and that we will be briefly explaining in the next section.

Microsoft supplied us with tools to help carry out various MBT activities, and they established communication channels through which we were able to report bugs, request development support, and resolve conflicts and ambiguities. We started by gathering a team with a rich, varied background. They all had been receiving some sort of formal university education in the fields of computer sciences, software engineering, and mathematics for a while, and they were fairly distributed across educational levels from those just starting their undergraduate studies to those pursuing doctoral research. Four out of the five working on the project had received proper instruc-

tion in software testing and two had previous experience in applying the technique in other projects administered by the Center (Jorgensen, 2000).

Both product and project conditions seemed to be in favor of employing a finite state model based technique. First, all the products had graphical user interfaces and seemed to be state rich, making them ideal for modeling using finite state models according to earlier case studies. Second, the products were in general small; they had significantly less inputs and features than what one would expect in a similar desktop applications. We believed, at first sight, that the environment in which the Pocket PC application were deployed to be relatively well behaved. For instance, there were only a few other applications with which they would interact, and most of these interactions could be manageably monitored and recorded. In addition, by virtue of its design, which was intended to support only a few devices and did not have any backward compatibility issues, the operating system was small, free of clutter functions, and well tweaked for its purposes. We had better chances, therefore, to accurately configure our tests and account for most environmental conditions.

The project spanned two academic semesters, which would amount to eight effective months of testing, more than many groups in the industry world normally have to develop a product never mind test it. All members were contractually obligated to work for at least twenty hours every week, but many ended up devoting up to thirty-five hours to this project. Given the fact that we were supposed to test five applications, however, this meant that we had just enough time, but not a whole lot.

We did have some worries about a number of practical issues, most notably input simulation and test outcome evaluation. We were not exactly clear on how to execute our tests in an embedded system; typically, in such cases, some type of simulator would be needed. As to evaluation, we were also not clear on how to verify the state of the application against our models and how to monitor and record any other needed application information. Both these concerns were addressed and resolved through development and test-tool support from Microsoft, details of which could not be disclosed as per our legal obligations toward the company.

1.3 This Paper

Working on this project was rewarding in terms of the lessons learned. Our experience reinforced some of the common beliefs about some of the benefits of model-based testing. On the other hand, many questions about the returns on investment, bug count, and model-adaptability to specific contexts were raised with no satisfying answer.

This paper summarizes this experience. First, we briefly visit our technique for finite state model based software testing. Then, we describe the applications under test and briefly outline the plan that we followed to test each of them. A summary of the proceedings of the testing effort is followed by a list of some the lessons learned.

2. Background

2.1 Definitions and Terminology

2.1.1 Software States

A software state is loosely defined as a condition of the software in which a certain collection of inputs can be applied. For example, consider a typical combination safe. For our purpose, let us

say we walk into a room with that has such a safe. Consider two general states that the safe can be in:

1. All tumblers are aligned and we can turn the handle to open the safe.
2. The correct sequence has not been applied to the combination dial and the safe handle can not be turned.

Here we have clear criteria to define states in terms of applicable inputs. One state is defined by the fact that we can turn the handle and open the safe. In the other state this input is not available to us. We can easily extend this criterion for state definition to software. Consider a typical GUI email application. In most such applications if there is no entry in the "To" line the send option is disabled. Intuitively we can say that when text is in the "To" line, the software is in one state which is different from the state the software is in when text is not present because different inputs are available to the user. In the following sections we will cement this notion of a software state through examples and formal definition.

2.1.2 An Example Software Under Test

Consider a hypothetical light switch. 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. Obviously, the simulator can be in only one of four distinct states at any one time: the lights are either off, dim, normal, or bright.

2.1.3 Finite State Machines

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*.

2.1.4 Example Finite State Machine

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 con-
            dition not described above
o   F = [off]
o   L = [off]
```

Figure 1: "Light Switch" Finite State Machine Definition

2.1.5 REPRESENTATION

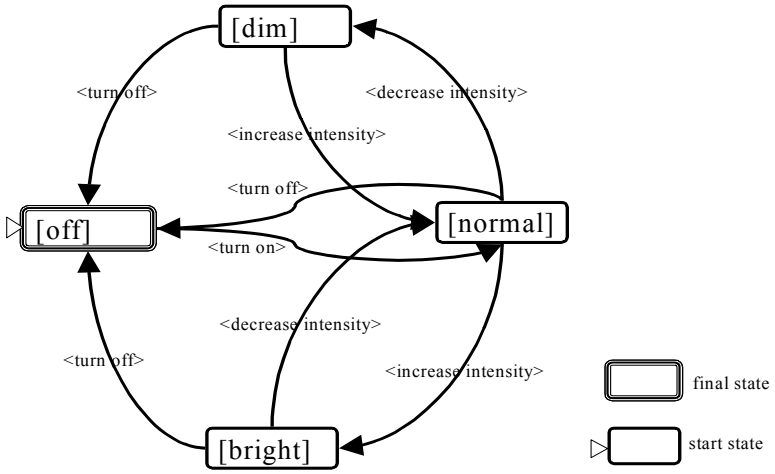
Finite state machine models can be represented as graphs, also called state transition diagrams, with nodes representing states, arcs representing transitions, and arc-labels representing inputs causing the transitions. Usually, the starting and final states are specially marked. Automata can also be 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>	

(i)

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

(ii)



(iii)

Figure 2: The presentation screen of the inbox application

2.2 Why Finite State Models are Useful

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 from which to choose. 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 from which testers can select. If one accepts this description of software then a model that must be considered is the *finite state machine*.

Finite state machines 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. Chow (1978) wrote 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 good models for describing sequences of inputs. This, combined with a wealth of graph traversal algorithms (Robinson 1999 TCS), 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.

2.3 Finite State Model-Based Testing Activities

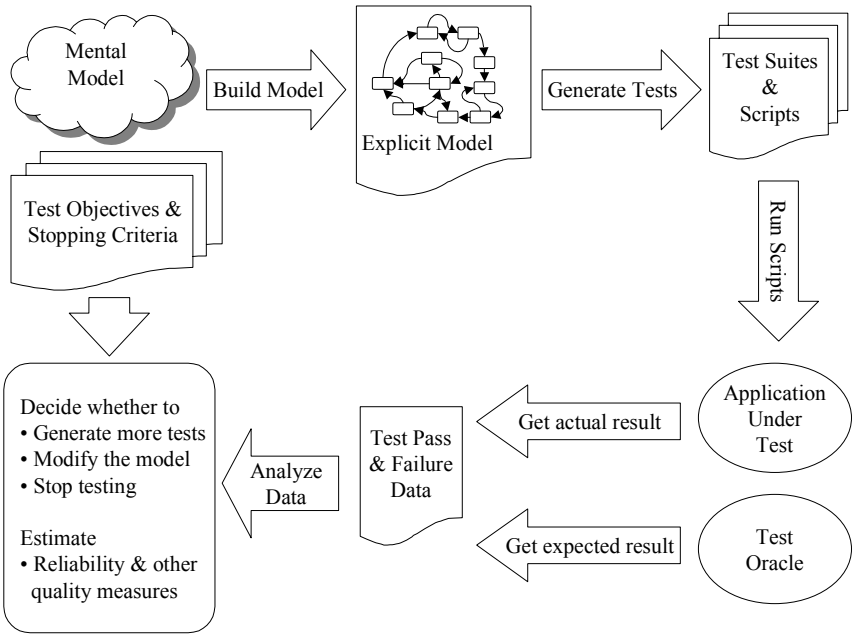


Figure 3: Some Model-based Testing Activities

Figure 3 above describes the finite state model based testing process. Perhaps the most difficult step is encapsulating our mental model of the software into a concrete structure. In the next section, we discuss a framework for expressing software models and representing states as a collection of software attributes.

2.4 A Compact Representation of Finite State Models

Directed graphs representing the functionality of a software component can be an effective tool in software testing. Figure 2 shows an example of such a graph. For any software system of non-trivial size, however, these representations are inadequate. Here we seek to define a compact way of defining a software state in terms of critical characteristics of the software. For example, consider the email application described in section 2.1.1 above. Recall that the "Send" button of the

application is disabled if no text is entered in the "To" field. Suppose this application has two windows: the first, which lists all received email and the second, which allows the user to compose an email. Here we can identify two critical conditions that must be met in order to apply the "Send" input:

1. The application must be in the "compose" window.
2. Text must be present in the "To" field.

To encapsulate this information, each of these characteristics of the system is referred to as a state variable. For example, for this application we may want to define our state variables as Window and Text_in_To_Field. Associated with each state variable is a set of values. In this case it would be appropriate to define:

Window = View, Compose
Text_in_To_Field = Yes, No

A state then in terms of these values can be thought of as the combination of the variables above with one value for each. The only state in this case for which we can apply the "Send" input is:
{Window = Compose and Text_in_To_Field = Yes}
The total number of potential states is the cross-product of the number of state variable values. In our example the total number of possible states is thus 4 (2* 2) because we have 2 values for each state variable. However, the number of valid states is almost never equal to this total. This is one of the problems of model-based testing in that a significant amount of time is generally spent identifying impossible states. In this example, there are only 3 possible states, which are:
{Window = View and Text_in_To_Field = No}
{Window = Compose and Text_in_To_Field = No}
{Window = Compose and Text_in_To_Field = Yes}

Another significant issue in model-based testing is state explosion. State explosion generally happens when we increase the number of state variables and/or values. Consequently, adding only one value can result in an out-of-control number of valid states, especially for large models. Consider in this case adding just one value to the Window state variable. This action will increase the number of potential states from 4 to 6.

3. The Testing Effort: The Inbox

The Inbox application is a small-scale version of outlook. It is the largest application we had to test. The whole model consisted of almost 5,000 transitions and approximately 1,500 states. This application was interesting to model, in that it had a diverse range of features. An example of this diversity is the number of different windows in the application. The user can either view the inbox, outbox, deleted-items or drafts. One can edit a message from all these screens except from the deleted-items window. To limit state explosion, we had to limit the number of messages that could be found in each of the windows to three(except the deleted items window).

We encountered some interesting challenges while modeling this application and automating tests. For example, we actually needed to start running test suites with the keyboard visible on the screen to ensure that inputs were accessible by automation. Such problems are often met while using MBT techniques and these design choices that the developer makes in order to be user-friendly sometimes force the testers to model around them.

Most of the defects we found were in this application. This is understandable as it was the most complex and the one with the most features modeled. This also confirms our intuition that modeling more details often increases the chances of finding defects. One has to be

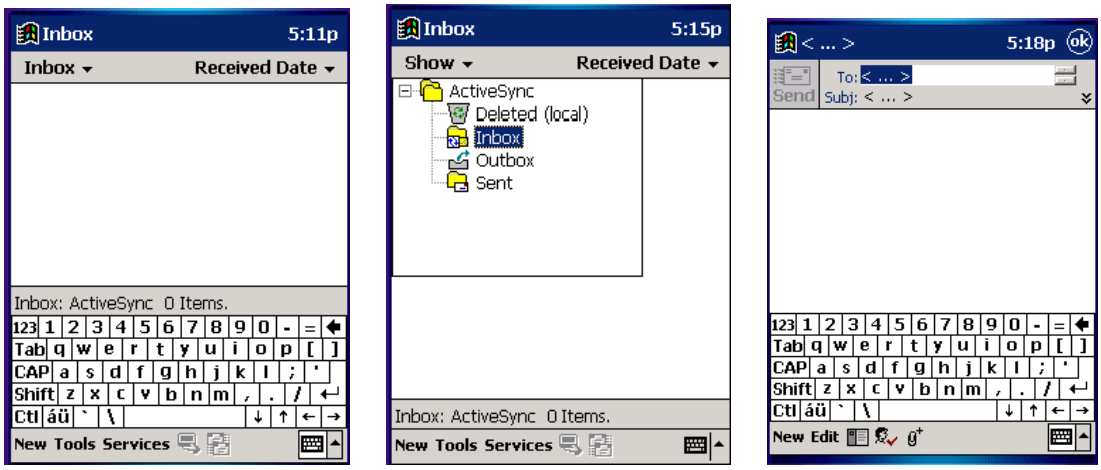
careful, though, as one potential pitfall when using MBT techniques is to construct an overly detailed model. Such a model is seldom readable, hard to maintain, and makes building automation much more difficult.

Next, we present the steps necessary for our finite state model-based technique. For each step, we use examples from our work on the inbox application. The complete model would be confusing as it is too big; thus, we show only a partial model.

1. Explore the application in order to discover and build a mental representation of its functionality.

   For the Inbox application, we first studied the different screens[1].

   Figure 4(a), shows the first screen that users see when they enter the inbox application. The second screen, figure 4(b), exposes the menu that allows a user to browse through the inbox folders. Figure 4(c), shows the new message screen that appears after the user clicks on "New". On this screen, the user can type a message and send it. The "Send" button will only be enabled when the user has typed some text in the "To" field. To save a message the user will have to click the "Ok" button.



(a) (b) (c)

Figure 4: Different Screens of the Inbox Application: (a) Presentation Screen of the Inbox Application (b) Presentation Screen and the Inbox Menu (c) New Message Window

2. Identify all user inputs. Decisions on whether to abstract physical inputs are made based on what we need to test. For example, two inputs that are visibly the same and that can be simulated with the same script may be abstracted as one input.

   While we were testing the inbox application, we made a lot of abstractions. Figure5 shows a screenshot of the New Message window. The "Ok" input located in the upper-left corner of the window has the same effect as pressing the Enter button. Since we could not find any

[1] The screenshots in this paper were captured from the publicly available Pocket PC emulator. The actual (similar) beta versions we worked with fall under non-disclosure agreement.

significant differences between these two inputs, we decided to only consider "Ok" as an input and not Enter.

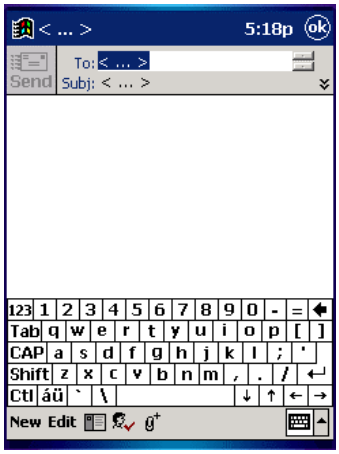


Figure 5: Abstraction of Inputs

Here is the full list of inputs with short explanation for each of them for the small model presented in this section.

1) New: to go to a new message window
2) Ok: to leave the New Message screen and save the current message
3) Send: to leave the New Message screen and send the current message
4) Space: space character
5) AlphanumericChar: any alphanumeric character

3. Identify the individual characteristics used to define the states of the application. When can this input be applied by the user and what are the system characteristics that affect its applicability? What are the properties of the system that cause different responses to the same input under seemingly similar conditions? From this information, define the rules that describe valid sequences of inputs.

   For each model, we explored the application in more depth than in step 1 to uncover input's applicability. Next, we describe the conditions in which each input is applicable.

   1) New: this input is applicable at any time. The user can press the "New" button when the general inbox screen is showing, when the window is New Message and whether he/she has entered text inside a New Message window.

   2) Ok: this input is applicable when the window is a New Message. Whether the "To" field is empty or not does not make a difference.

   3) Send: this input is applicable only when the window is a New Message and the "To" field is not empty.

   4) Space: this input is applicable when the window is a New Message.

   5) AlphanumericChar: this input is applicable when the window is a New Message.

Applicability of inputs then allows the tester to derive valid sequences of inputs. For our model and assuming that the starting state is the general inbox window, an example of a valid sequence is: New-AphanumericChar-Space-Send.

4. Generate the set of valid software states and transitions with the aid of specialized tools.

   Following is the list of operational modes and inputs that are necessary in order to construct our scaled down model of the inbox. Its state transition diagram will then be shown.

   State variables

   Window = Inbox, NewMessage
   This operational mode records which window the user is on. For this model, only two windows are considered as possible.

   To Field = Empty, NotEmpty
   This operational mode records if the value of the "To" field is empty or not. It is useful to determine whether the "Send" is enabled or not.

   State transition diagram

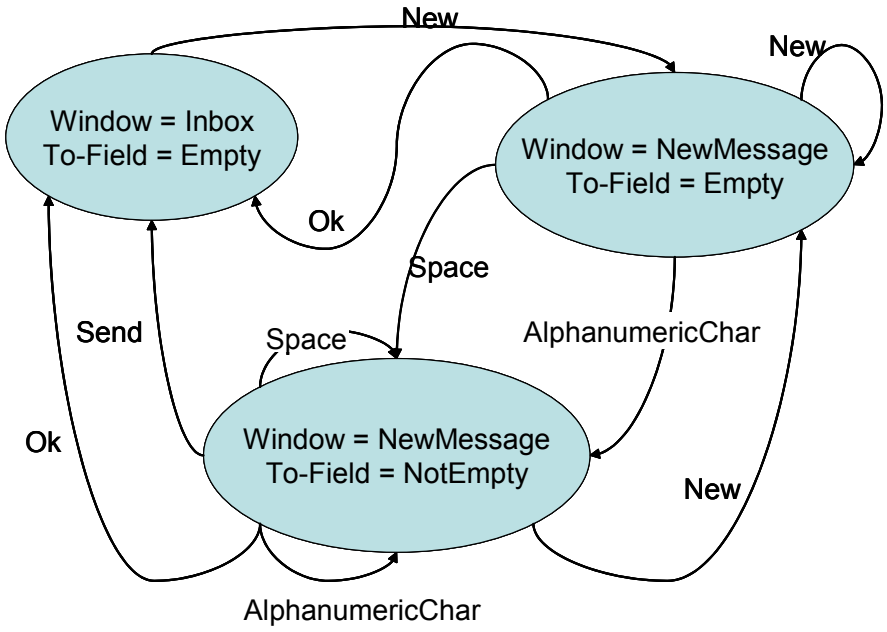


Figure 6: Graph Representation of the Model

Examples of input sequences are New-Space-Ok, New-New-Ok etc.

5. Generate and run test cases (traversal paths in the graph). This step particularly benefits from the well-established graph-theoretical body of knowledge.
   To generate and run test cases, we used tools provided by Microsoft that we cannot disclose in this paper. However, to better understand some of the sequences in this model, the follow-

ing shows one such possible scenario. Figure 8 below shows the path that is illustrated through the screen shots below in figure 9.

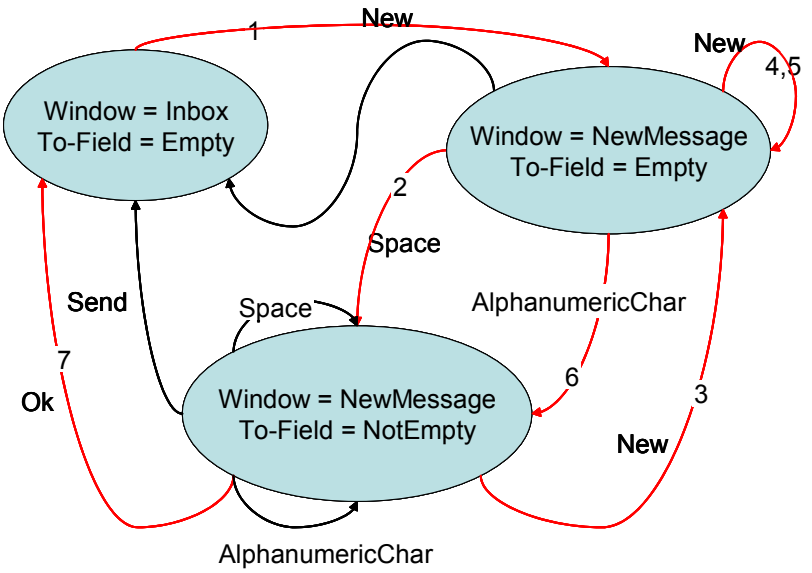


Figure 7: Traversal Path

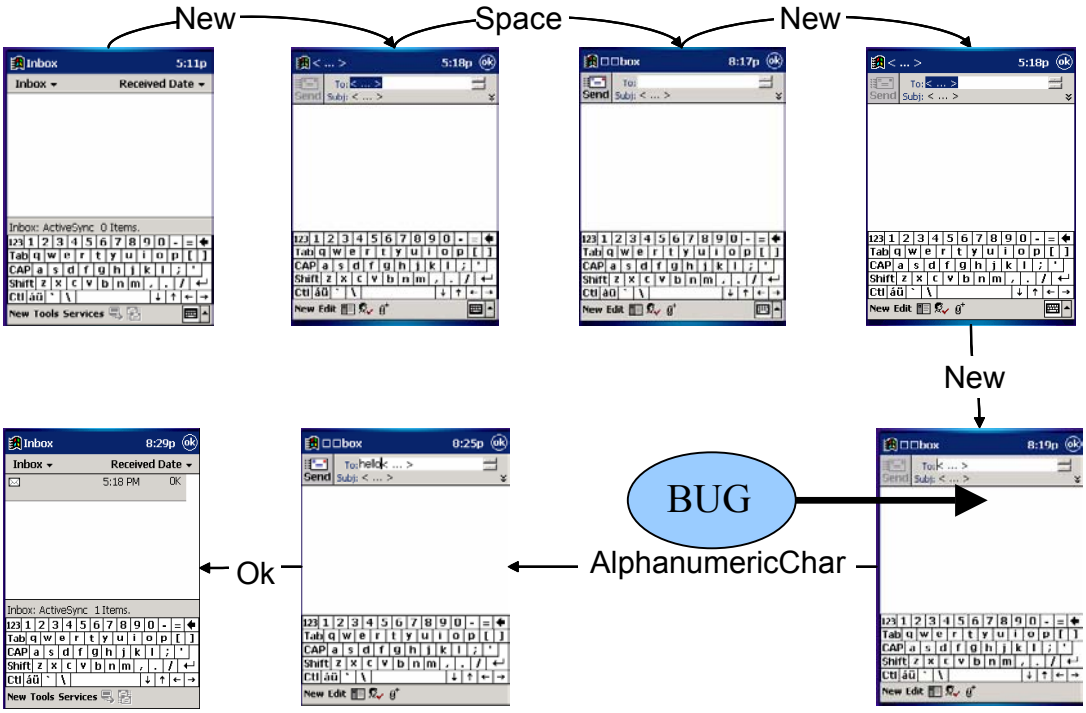


Figure 8: Scenario in Screenshots

This series of screenshots is an example of test sequences that were run. This sequence also demonstrates an inconsistency in one of the inputs. The first two "New" inputs show a New

Message screen with the "To" field highlighted. This allows the user to type text directly into the "To" field. The third "New" though shows a New Message with the cursor in front of the "< … >". It means that the user will type text in front of that default value, which will stay unless the user manually removes it (see the screenshot resulting from the "AlphanumericChar" input). This issue was accepted as a defect by Microsoft and even if it seems like a small problem, it would still inconvenience the user. The Inbox application was the largest application we had to test and we found a number of inconsistencies that were accepted as defects. Uncovering such failures illustrates an advantage of model-based testing techniques over other testing techniques; sequences of inputs that are unusual and do not seem to be potentially defective are executed by finite state machines.

4. Conclusions

- Model-based testing needs to be coupled with exploratory techniques with the dual benefit of attaining a better, more current understanding of the system and harvesting many bugs along the way.
- Models are beneficial, not only as a point of reference for testing purposes, but also as a living specification of the functionality it represents and as a basis for test automation.
- Having a good automated test oracle is vital to the effectiveness of automated testing in general and model-based testing in particular.
- As long as finite state machines are used, there are inescapable critical issues to be dealt with: model building and maintenance, state explosion, and model correctness. There is need for more practical pointers on how to work around, or at least reduce the impact of these factors.
- Finally, studies need to be performed on answering the question: is model-based testing worth the effort when it comes to finding faults? Our preliminary results show that, for a very good, close-to-release, stable product, the number of faults uncovered by model-based testing is slightly disappointing if we severely limit the time during which we can run tests. The strongpoint of model-based testing is that it finds bugs with different characteristics: those that require long complicated sequences of inputs to be exposed.

Acknowledgements

This project was sponsored in part by a grant from Microsoft Inc in 2000. Thanks are due to Ahmed Salem, Matt Wagner, Danko Panic, and Mohammed Al-Ghafees for their work and contribution to the project.

About the Authors

Ibrahim K. El-Far is a doctoral candidate in computer sciences at the Florida Institute of Technology under the academic supervision of professor James A. Whittaker. He has a Bachelor of Sciences and a Master of Sciences in Computer Science from the American University of Beirut, Beirut, Lebanon and the Florida Institute of Technology, Melbourne, Florida, USA, respectively. His interests are in investigating software models for testing, test automation and tools, adequacy criteria, test cost and effectiveness, and software testing education. In 2000, Ibrahim received a fellowship from the International Business Machines Center for Advanced Studies, Canada, supporting his research in approaches to interpreting, understanding, and analyzing software test effectiveness and efficiency. He has over four years of experience in model-based testing using finite state machines at the Center for Software Engineering Research at Florida Tech, where he has supervised the development of experimental model-based testing tools, advised model-based testing groups, and taught model-based testing in various formats to a variety of students. You

can contact Ibrahim at ielfar@acm.org, visit http://www.testingresearch.com/ for more information, or write him at the Software Engineering Program, Computer Sciences Department, Florida Institute of Technology, 150 West University Boulevard, Melbourne, Florida 32901 USA.

Herbert H. Thompson is a doctoral student in mathematics at the Florida Institute of Technology. In the past, he has worked for Microsoft Corporation as a test engineer. His research interests are in software engineering, security, and applying mathematics to computer science problems.

Florence E. Mottay is a graduate student in software engineering and a research assistant at the Center for Software Engineering Research, Florida Institute of Technology, Melbourne. Her research interests are in software testing, formal languages, mathematical models, and e-commerce. She received awards for excellence in mathematics by the United States Achievement Academy (1997) and for academic excellence by the American Association of University Women (1998).

References

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

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

[Chow 1978] Tsun S. Chow. Testing design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3): 178-187, May 1978.

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

[El-Far 2001] Ibrahim K. El-Far. Enjoying the perks of model-based testing. *Proceedings of 2001 Software Testing Analysis & Review Conference (STARWEST)*, San Jose, California, USA, October/November 2001.

[Gross & Yellen 1998] Jonathan Gross and Jay Yellen. *Graph theory and its applications*. CRC, Boca Raton, FL, USA, 1998.

[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.

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

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

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

[Robinson 2000] Harry Robinson. Intelligent test automation. *Software Testing Quality Engineering (STQE) Magazine*, October/November 2000.

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