Model-Based Testing of Installers in a Development Test Environment

Chris Struble
Hewlett Packard Company
11311 Chinden Blvd.
Boise, Idaho USA
83714-0021
chris.struble@hp.com

Abstract
Model-based testing is an evolving technique for generating test cases automatically from a behavioral model of a system. This technique was applied to software testing of print driver installers in two case studies in a development testing organization. The challenge was to develop a suite of tests for new installer software. The suite had to provide thorough coverage in a reasonable number of tests, be easy to modify and leverage to new projects, and be compatible with existing test management and automation tools and other standards of the testing organization.

The software was modeled as an extended finite state machine. An initial test suite was created from a state diagram drawn by hand from specifications and exploratory testing. A second test suite was developed using TestMaster, a modeling and test generation tool. Test cases for manual execution and automated execution were generated. Model-based testing worked very well for installer test development, and the test suites proved very effective at finding defects. Advanced modeling techniques were also investigated and evaluated for practicability. Learnings, best practices, and problems remaining to be solved are discussed.

Keywords
Model-based testing, test generation, installer testing, test development, test maintenance

1. Introduction

Model-based testing is a technique for automatically generating test cases from a model of the behavior of the system under test. Models are used to understand and specify complex systems in many engineering disciplines, from aircraft design to object-oriented software development. Use of models to generate test cases is widely used in telecommunications (for example, see [Clarke 1998]) but is relatively new in software testing. Interest in using models to generate software tests is growing for many reasons:

- Models provide a formal way to document, share, and reuse information and assumptions about the behavior of software among test developers and software developers.
- Models can be fed into a test case generator to generate a suite of test cases automatically. Models thus offer the potential to develop test suites more quickly than manual test generation methods.
- Model-based test generation algorithms can guarantee coverage based on constraints and other criteria specified by the test developer.
- The process of creating a model of the software can increase the test developer's understanding of the software and can lead to better test cases even if an automated test case generator is not used.
- Models can be copied and modified to quickly produce models for related software products, especially software products based on common source code.
- Models can be easily and quickly updated when the software changes, and used to quickly re-generate test suites that are consistent with the current correct expected behavior of the software.

All of these are good reasons to consider model-based testing, but it is the last two that can provide the most significant benefit for software testing organizations in practice. Model-based testing is not only an efficient way to generate test cases; it can be a powerful tool for managing the problem of test maintenance. This turns out to be very important for installer testing.

2. Test maintenance

Test generation and maintenance is one of the most challenging problems facing software test organizations. Software complexity is increasing while software test schedules are shrinking. Test developers are tasked to generate very large test suites in short times, and to maintain them in good working order as software changes. Many organizations are also moving toward automated test execution to test more quickly and to reduce test execution costs. However, automated execution makes test suite maintenance even more critical, because automated test harnesses are very sensitive to errors in test scripts and prone to fail en masse at the slightest mismatch between the software and the test suite. At the same time, high quality standards must be maintained, requiring test suites that not only run without breaking, but also thoroughly cover the software behavior and expose as many defects as possible.

In most software test organizations, test suites are still generated and maintained by hand, in a process that is often tedious, repetitive, boring, error prone, undocumented, and not supportive of reuse. During the test generation phase, the test developer must pour over specification documents, prototypes, and other artifacts to try to understand how the software is intended to work. Based on that understanding, or internal mental model, the test developer must write or modify hundreds or thousands of test cases and verify them for correctness. For automated tests this means running them against a test harness and possibly a test oracle.

Almost immediately the software begins to change. For each software change, the test developer has to update the mental model, hunt down the test cases that need to be changed, apply the changes, and verify them for correctness. As the software becomes more complex, the number of tests that need to be searched increases, the number that have to be identified and changed increases, and identifying them becomes more difficult because software changes impact test cases in increasingly complex ways. And if the test developer leaves the organization, his successor must repeat the entire process because the original developer's understanding of the software behavior was never documented directly.

Eventually the test developer gets overwhelmed. The test suite and the software get out of sync, many test cases break and have to be taken out of production, important software behaviors are never tested, and defects escape detection. This is the "test maintenance problem". It is a real phenomenon the author has experienced in practice. This anecdote from [Mosley, 1997] is revealing:

> *"I worked with one Visual Basic programmer who had previously constructed and maintained an automated suite of test cases for a client-server system. The problem he encountered was that the features were never frozen and he was constantly updating the test scripts to keep up with all of the changes. Of course, he never caught up. He swears he will never do testing again!"*

What is needed is a way to reduce the points of maintenance, and the time it takes to do test maintenance, by orders of magnitude. Model-based testing offers just that. To illustrate this we will examine a particular type of software model in detail, the extended finite state machine (EFSM) model, using installers as an example.

3. The extended finite state machine (EFSM) model

Many types of models can be used to represent software behavior (see [El-Far and Whittaker 2001] for a description of several techniques), but one of the most widely used is the finite state machine, or FSM.

Mathematically, a simple finite state machine (or finite automata) is represented as a 5-Tuple $(Q, \Sigma, \delta, q_0, F)$ [Sipser, 1997] where:

1. Q is a finite set called the **states**;
2. Σ is a finite set called the **input alphabet**;
3. δ: Q x $\Sigma \rightarrow$ Q is the **transition function**;
4. $q_0 \in$ Q is the **start state**;
5. F $\subseteq$ Q is the set of **accept states**.

More intuitively, an FSM can be represented graphically, with the set of states Q represented as a set of circles, and the set of transitions δ represented as a set of directed lines connecting the states. Each transition is labeled by the input that causes that transition to be taken, moving the machine to a new state.

For modeling interactive software such as an installation program, states in an FSM should represent the information that is currently being presented to the user (the screen or dialog currently displayed, or its appearance and/or settings), while transitions represent user actions (the user clicking on a button to cause a dialog to appear) or actions performed by the software itself (such as copying files).

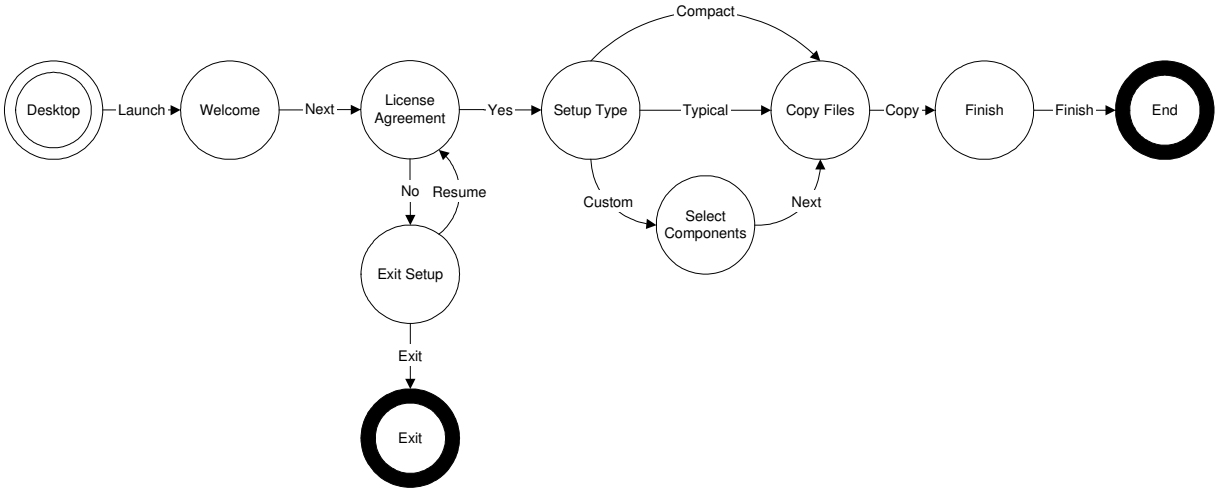

Figure 1 – FSM for a simple installation program

Figure 1 shows an example of a simple installation program modeled as an FSM. Each state has a name describing the dialog or other information currently presented to the user. Each transition has a name describing the action causing the software to move along that transition to a new state. The model has an initial state called Desktop and two stop states, Exit and End.

A **path** is a unique sequence of states and transitions through the state machine model. Consider the following path through Figure 1:

Desktop → Launch → Welcome → Next → License Agreement → No → Exit Setup → Exit → Exit

What is happening here? In this example, the path begins with the Windows desktop visible (the Desktop state); the user launches the installation program (the Launch transition); the Welcome dialog is displayed; the user clicks the Next button; the License Agreement dialog is displayed; the user (apparently having read the license agreement and found it to not be agreeable) clicks the No button; the Exit Setup dialog is displayed

with the text “Are you sure you want to Exit?”; the user clicks Exit; the installation program exits and the Windows desktop is visible once again (the Exit state).

Suppose we wanted to create a test procedure for manual execution, in English, corresponding to the above path through the installation program. It might look something like this:

1. Verify that the Setup program icon is visible on the Windows desktop
2. Double click the Setup program icon to launch the Setup program
3. Verify that the Welcome dialog is displayed and the Next button has the focus
4. Click the Next button
5. Verify that the License Agreement dialog appears and that the Yes button has the focus
6. Click the No button
7. Verify that the Exit Setup dialog appears and that the Resume button has the focus
8. Click the Exit button
9. Verify that the Setup program exits and the Windows desktop appears

Note that while there are only four transitions in this path, there are nine lines in the procedure. The first three lines correspond to the first transition (Launch), lines 4-5 correspond to the Next transition, and so on. Thus each transition corresponds to a segment of zero or more lines of **test code**, and the concatenation of all the lines of test code collected from the transitions along the path make up the test procedure. Ideally we would like to attach this test code to the machine somehow so that when a path was traversed through the machine, a corresponding procedure could be generated automatically.

The machine in Figure 1 has some problems, however. First, from the definition of the simple finite state machine, there is no obvious way to attach the test code to the transitions in the machine. The simple FSM doesn’t allow for this. The capabilities of the FSM need to be extended to permit attaching test code to transitions.

Second, there is a problem with the Copy Files state and the Copy transition. In the Setup Type dialog, shown in simplified form in Figure 2, there is a button group with three options: Typical, Compact, and Custom. These refer to **setup types**, a concept in installers meaning that if the user selects that type and clicks Next, a specific set of files will be copied to the user’s computer.

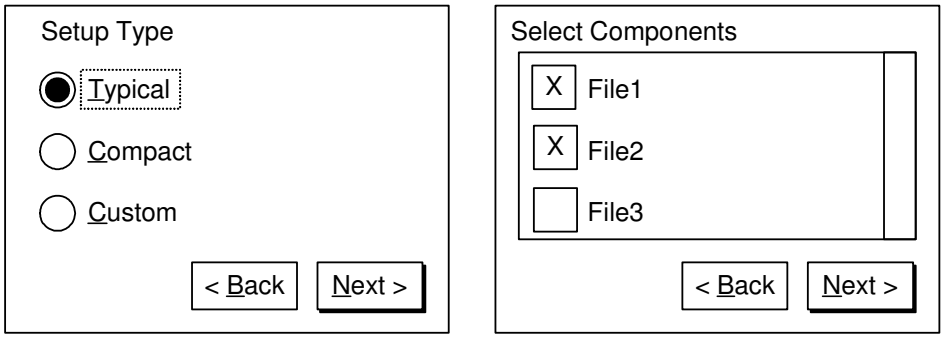

Figure 2 – Setup Type and Select Components installer dialogs

Let us suppose the installer in Figure 1 can copy three files, File1, File2, and File 3. Suppose further that Compact setup installs File 1, Typical setup installs File 1 and File 2, and that Custom setup will display a

Select Components dialog and allow the user to select any combination of File1, File2, and File3 so long as at least one file is selected.

Ultimately the path ends up in the Copy Files state, and in the Copy transition the installer begins copying files. But which files? How does the Copy transition know which set of files to copy? It doesn't. Simple FSMs don't have the ability to store data such as the setup type or list of files the user selected earlier in the path. The capabilities of the FSM need to be extended to allow modifying, storing, and accessing data.

Figure 1 has another problem. To find it, consider the question "How many possible paths are there through this model?" This is another way of saying, "how many unique test cases can I generate?" In this case, the number is infinite. Why? Because between the License Agreement state and the Exit Setup state is a loop. The pair of transitions No and Resume between these states can be looped any number of times, with each additional loop creating a unique path.

Clearly we need a way to limit the number of times a test will traverse a loop, or any transition in general. We need to be able to express rules such as "don't allow this transition to be taken more than 3 times in any path" or "don't allow this transition to be taken more than once in the entire test suite". These are not limits imposed by the software behavior, but by test coverage criteria.

Sometimes it is also necessary to impose rules on the model based on the software behavior itself. Suppose that we consider our installer example from Figure 1. We would now like it to represent an installer that installs print driver files, and sets up a connection to a printer so the user can print to it after the installation is completed.

Suppose that the software is changed to allow the user to select the type of printer they want to connect to, either a network printer or a local printer connected directly to the user's computer. We want the installer to detect if the user's computer has a network connection, and to only ask the user to select a connection type if the user's computer has a network connection. If the user's computer does not have a network connection, we don't want to ask the user to select a connection type, because only one connection type (local printer) is possible in that case.

Suppose that a dialog called Connection is added to the installation sequence (just before the Setup Type dialog) to allow the user to select the connection type. This results in a model such as shown in Figure 3.

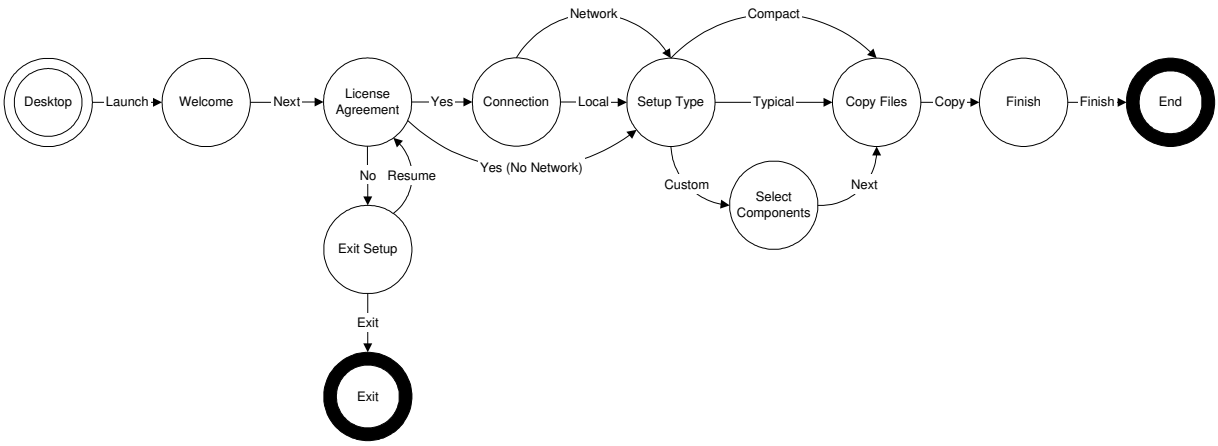

Figure 3 – Installation program with added dialog

In Figure 3 we have added the Connection state to represent the Connection dialog being displayed. We have also added two transitions, Network and Local, from the Connection state to the Setup Type state. These represent the user selecting either the network printer option or local printer option and clicking Next on the Connection dialog. There is also a new transition from License Agreement to Setup Type called Yes (No Network). This represents the transition taken if the user clicks Yes on the License Agreement dialog and the user's computer does not have a network connection. The Connection dialog is skipped entirely in this case. The Yes transition from the License Agreement dialog has also been changed to connect to the Connection state rather than the Setup Type state. This represents the transition taken if the user clicks Yes on the License Agreement screen and their computer does have a network connection.

To make this model work correctly, we need to be able to attach the constraint "Only allow this transition to be taken if the user's computer has a network connection" to the Yes transition coming out of License Agreement. We also need to be able to attach the constraint "Only allow this transition to be taken if the user's computer does not have a network connection" to the Yes (No Network) transition. Again, the capabilities of the FSM need to be extended to handle such constraints.

To summarize, to model real software, the FSM needs to be extended to include the following capabilities:

- Ability to attach test code to objects in the state machine
- Ability to handle context (remember path taken so far)
- Ability to store, modify, and access data variables
- Ability to express constraints on transitions to block invalid paths

TestMaster, the graphical modeling and automated test generation tool used in the case studies, supports all of these concepts. However, the concepts are more general and need to be considered when using any tool.

The author promised to illustrate how model-based testing could help reduce the time needed to do test maintenance. Using only the diagram in Figure 1 as a reference, the extended finite state machine (EFSM) for the simple installer in Figure 1 was modeled in TestMaster. This took 45 minutes, including test code in English, necessary data variables, test generation, and debugging time. This process produced 10 test cases totaling 146 lines of test code in full cover (all possible paths). A screenshot of the TestMaster model is shown in Figure 4.

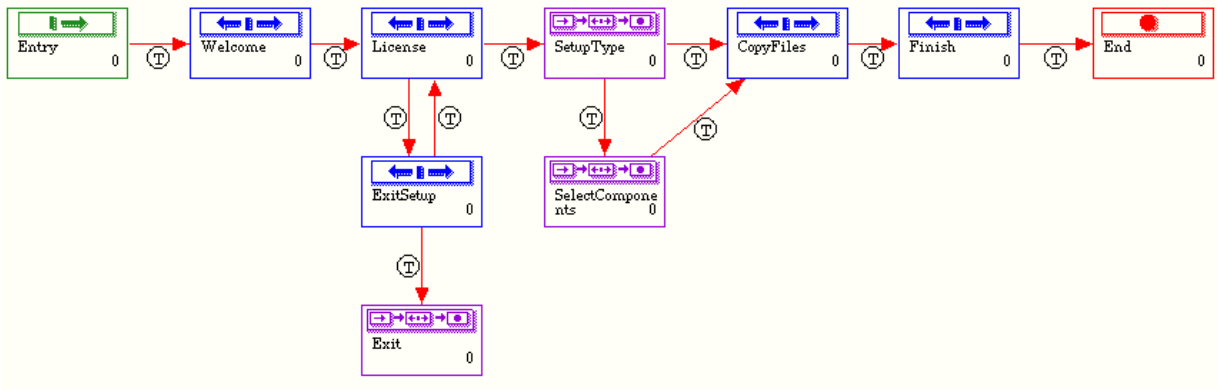

Figure 4 – TestMaster model for the simple installer in Figure 1.

Here is one of the test cases produced by the model in Figure 4.

Verify that the Setup program icon is visible on the Windows desktop
Double click the Setup program icon to launch the Setup program
Verify that the Welcome dialog is displayed and the Next button has the focus
Click the Next button
Verify that the License Agreement dialog appears and that the Yes button has the focus
Click the Yes button
Verify that the Setup Type dialog appears and that Typical is selected and Next has the focus
Click the Next button
Wait while files are copied
Verify that the Finish dialog appears
Click Finish
Verify that the Setup program exits, that file1 and file2 are installed
Verify that the local printer can print a test page

Adding the states, transitions, test code, and other data to transform the model in Figure 4 to represent the state machine in Figure 3, took only 15 minutes including test generation and debugging. TestMaster generated 26 test cases totaling 468 lines of test code for the new model in full cover. Modifying the original 10 test cases into the 26 new test cases by hand would probably have taken several hours, not only because of the new test cases, but because almost all of the existing test cases would have to be changed to accommodate the new dialog into the sequence.

The previous example illustrates that once a model is generated, making a small change to it (such as adding a dialog) takes very little time, but may result in a large change to the test suite. This holds true even for very large models. A large model will take longer to regenerate the tests than a smaller model, and for larger models it may take longer to track down an error. But even for large models, most of the actual time is spent in modeling and re-modeling, not test generation or debugging. This leads to the following observations:

- For EFSM model-based test generation, the amount of time and effort needed to apply a software specification change to a model depends strongly on the size of the change but only weakly on the size of the model.
- For manual test generation, the amount of time and effort needed to apply a software specification change to a suite of test cases depends strongly on both the size of the test suite and the size of the change.

It should be noted that test suites are more sensitive to software changes for some classes of software than for others. Installers are an example of "sequential" software in that the user generally has to pass through a long sequence of states to execute a useful function (e.g. copying files). Wizards are another example of this type of software. For sequential software, many paths will pass through the same state, so a small software change, especially early in the sequence, will affect many paths and break many tests. Model-based test generation provides real benefit here because the maintenance task is reduced from the size of the test suite for manual test generation, to the size of the change in the model.

4. Case study: Manual test generation from a state machine model

Two testing projects are offered as case studies to show how model-based testing can be applied to installer testing in practice. Both involved developing a new test suite for a common installer architecture for installing HP printer drivers on Windows operating systems. The architecture was designed to support a variety of printer products and to be customizable, so it contained a superset of features some of which would be used by some printer products and not others. Among the features were express, typical, and custom installations; support for network, USB, and direct printer connections; installing files from CD, network, or the web; setting up the printer in the user's Printers folder, sharing the installed printer on the network, and registering

the product with HP over the internet. Also included was a utility that would allow network administrators to create customized installers with some options pre-selected for users. The software had to be tested on several operating systems, over many network configurations, in several languages, in less than six months and on a modest testing budget. Existing installer test suites could not be used because the architecture was completely new source code.

In the first case study, a test suite was developed using a hybrid approach. This involved creating a state machine model and generating the test cases from the model manually. This was done primarily because of schedule and because the author had not been trained on TestMaster at that time. Working from user-level specifications and early releases of the software, the author created a modified finite state machine diagram of the installer with Visio™, a drawing application. Each dialog in the installer was modeled as a rectangular box (state) in the diagram. Each button or action that would enter or leave the dialog was also modeled as a transition arrow. Within the state box were the name of the dialog and every feature of the dialog that had to be tested. These included:

- Verifying that the text on the dialog was correct (once per dialog)
- Verifying the tab order and that every control was accessible by tab key (once per dialog)
- Verifying hot key and arrow keys worked correctly (once per dialog)
- Canceling from the installation (once per dialog)
- Each other feature of the dialog had to be used in a complete installation (i.e. in a path that did not result in canceling from the installation) at least once.

Once the diagram was completed, the test cases for manual execution were written one at a time while tracing through the diagram step by step, marking with a highlighter each feature and each transition that had been covered by that test case that had not been covered yet by previously written tests. As each test case was written, it was verified on an early software prototype. The process was continued until every feature row and every transition arrow in the model was marked off. Figure5 shows a diagram similar to that used in the case study.

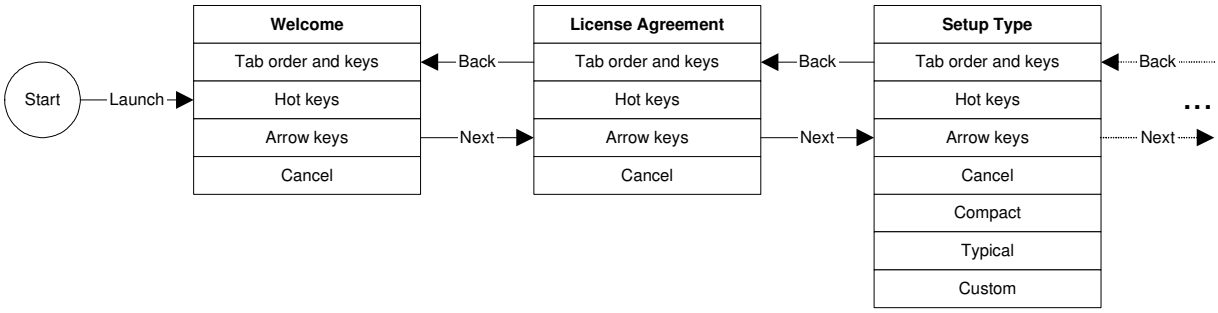

Figure 5 – Part of a modified EFSM model for use in manual test generation

Creating the model and writing the test cases from the model took about eight weeks. About 300 manual test cases were generated from this process, with an average of about 15 lines per test case. The process took slightly longer than manual test development from specifications alone, but produced a much more thorough set of test cases. The model also helped to ensure coverage, and the process of modeling increased the test developer's understanding of the software, raised questions that might not have come up otherwise, and generated many defects early in the development process that were also fixed early.

For the project overall, 300 test cases were run on an average of 3 operating systems each, detecting 100 distinct defects in the software, the vast majority of which were fixed prior to release of the architecture to product development teams. This was an extremely positive result. While some of the success was due to early testing and a closer than usual working relationship with the development team, much of the credit belongs to the modeling approach.

A limitation of this approach becomes apparent during the test maintenance phase of the project. Because the test cases were generated manually, it was not practical to regenerate tests from the model as the software changed. The test developer still had to search through the test cases and apply changes manually for each software change. Still, the hybrid approach was beneficial because the state machine model helped produced a very good test suite as a starting point.

5. Case study: Automated test generation with TestMaster

After the completion of the first case study, test cases were developed for two printer products, the HP LaserJet™ 4100 and HP LaserJet™ 4550, which were planning to use installers based on the common architecture tested in the first case study. Only part of the architecture would be used for each of these products, so the scope was somewhat smaller than the first case study. TestMaster was used for this second case study. Goals for the project were to:

- Produce a test suite that covered every transition in the model at least once.
- Produce test cases compatible with standards required by the test management system and processes being used in the software test organization.
- Investigate advanced features of TestMaster, such as whether a single model can support multiple products (data-driven testing), and whether a single model could generate tests for both manual and automated execution.
- Investigate whether a test suite generated automatically by TestMaster could find defects not found with a manually generated test suite run on the same product.

The following were some of the modeling and test generation strategies that were used or tried:

- A separate model was created for each dialog in the installer. Each submodel had an entry state, a default state (the state the dialog would be in when it first appeared in the installation sequence), and an exit state. Any controls on the dialog were modeled at this level.
- The model hierarchy had three levels, starting with the models for each dialog at the lowest level, models that contained groups of dialogs close together in the installation sequence (e.g. all the models for the network installation path), and the overall (root) model at the top.
- Submodels were reused wherever possible. For example, each dialog has a Cancel button for exiting the program. A single Cancel model was used for all of these cases.
- Submodels were built one at a time, added to the overall model, and then tested for correct test output. This verified that each model worked correctly by itself and when integrated with other models. The overall model was thus built up in an iterative fashion.
- TestMaster's Path Flow Language was used to specify constraints to ensure that only valid paths were generated, and to manipulate data variables that determined test code output as well as paths.
- TestMaster provides two debuggers, a path debugger, and a test debugger. Both were used little in early modeling, but as the project grew in size and neared completion, it became necessary to use both to track down some errors.
- Table models were used to represent lists, list boxes, and checkbox or button groups such as the controls in Figure 2. A table model was also used at the front end of the model to represent product-specific data, with one row for the HP LaserJet™ 4100 and one for the HP LaserJet™ 4550.

One row would be enabled and the other disabled, forcing the path generator to generate tests for a one product or the other. This enabled a single root model to be used for both products.

- The test case standard required that each test procedure have "preconditions" at the beginning of the procedure that would specify the setup conditions necessary to run it. The Test Prefix feature of TestMaster was used to collect all the preconditions dynamically, based on the path taken by the path generator, and place them at the front of the procedure. For example, the precondition "PC has TCP/IP protocol installed" would appear only if the path generator took a network installation path. Otherwise it was not needed. This proved to be an elegant and efficient solution.
- TestMaster has a feature to create multiple output streams from the same model. This was used to create two output streams, one for manual test code, and one for automated test code. The same model was thus used to produce both manual and automated tests. In the event of failure of the automation system or an automated test, corresponding manual tests could be run.
- TestMaster has several coverage schemes used by its test generation algorithms, including: Quick Cover which generates a minimal set of tests quickly; Transition Cover which attempts to cover all transitions at least once; and Full Cover which looks for all possible paths. Another coverage scheme called N-Switch cover produces tunable levels of coverage between Transition Cover and Full Cover. One of these levels is Zero-Switch cover (N-Switch with N = 0), which produced coverage equivalent to Transition Cover. All coverage schemes were experimented with to understand their suitability for different purposes.
- Test output was in the form of a CSV file that could be imported into the test management system.

Not all of the strategies worked well. In some cases, a simpler approach worked better. For example:

- Modeling the full behavior of the Back button (allowing a user to back up in the installation sequence) proved difficult. Combinations of Back and Next would form complex loops that the path generator could not handle. Eventually it was decided to use simple loops and allow tests to have only one Back transition per path (so paths like Next – Next – Back – Back – Next could not occur). This approach worked well.
- The use of product-specific data to generate tests for different products with the same model proved to take a lot of work to implement in practice. The problem arose because each product had a different number of installable components, and different names for many components. A very complex model had to be built to handle the general case where each product could have a completely different set of installable components. I would not recommend using this generalized approach because it is easier to build a product specific model and then modify it for another product's feature set, than it is to build a general model that will work for all products.
- Quick Cover was found to work best during the model construction phase. It produced a reasonable number of tests quickly enough to support a rapid test-fix-test cycle on the model. It was planned to use Transition Cover for the final suite of tests, but for reasons unknown, Transition Cover did not work well with the installer model. It ran very slow and in most cases was unable to converge to a solution. Zero-Switch Cover produced full transition coverage much more quickly and reliably. Full Cover produced enormous numbers of tests, far more than could ever be executed, so it had no practical significance for this project.

The final model consisted of over 50 submodels, used over 80 data variables, and generated about 100 test cases with Quick Cover, 160 with Zero-Switch Cover, and over 10,000 with Full Cover. Test cases averaged about 20 lines. The entire effort took about two man-months. About half the time went toward trying out the more advanced modeling techniques.

A suite of 160 test cases for manual execution was generated for the HP LaserJet™ 4100 installer, imported into the test management system, and run on the product close to final release. The tests were run through several test sessions. After each session the testers would report errors in the test cases, the model would be

modified to fix the errors, and a new test suite would be generated within a day. The test suite found several new defects that had escaped detection during normal product testing using a test suite leveraged from the test suite generated manually that was used in the first case study. This was significant since both the old and new test suite were based on the same state machine model.

6. Integrating model-based test generation into the software test process

One of the challenges in taking model-based testing beyond the pilot stage is integrating it into established software test processes in organizations where model-based testing is being deployed. In the TestMaster case study, test cases were generated that met established test case standards, and which could be executed by testers or automated tools already present without retooling or retraining. But this only begins to address the integration challenge.

Model-based testing tools need to take into account the knowledge and skills of the people who will be using them. Successful use of tools like TestMaster requires users with a rare mix of skills. The user must have detailed knowledge of the product under test in order to create a model that correctly describes the software behavior. The user must have programming ability to properly describe constraints and to debug the model and the test cases it generates. Finally, the user must be an expert in the test execution language so that the test code generated is executable in the test environment where the test cases will be run. If model-based testing is to be deployed successfully, it must be in the context of processes that take into account these different domains of expertise, and that in most software organizations no one person can maintain expertise in all these domains.

Another challenge to successful integration is defining processes for using test generation in organizations where test management systems are also used. Test management systems are tools for managing large numbers of test case, typically in a database. Many software test organizations are moving to test management systems to bring the task of software test maintenance under control to some degree. It has been shown that test cases can be exported from TestMaster into TestExpert, a test management system, at test generation time, where test cases are based on requirements. [Gurel, 1999] But model-based test generation tools and test management systems have different paradigms built into them that aren't fully compatible, and which make them difficult to integrate in practice.

In the paradigm of model-based test generation, models are viewed as essential data to be maintained and reused, and test cases are viewed as a disposable commodity, to be regenerated at will. In the paradigm of test management systems, test cases are viewed as essential data, to be maintained in a database and reused. These paradigms come into conflict when the output of a test generator is stored in a test management system. For example, when the software changes, and a new set of test cases is generated, how do you determine if a particular test case is the same as a test case already in the database? Do you replace the existing test case with the newly generated test case, or do you give the new test case a new test case identifier? What if the existing test case has been run at least once? Is it appropriate to replace the old test procedure with the new one, when the test result really applies to the old procedure? What is it that makes two test procedures essentially the same, other than an exact text match?
One approach is to assign newly generated test cases new identifiers, allowing the test suite to grow over the life of a project as new tests are generated. This approach is unlikely to be accepted because widely used metrics such as "number of planned test cases" or "percentage of test cases executed" become meaningless. It also leads to suites with many test cases that are essentially duplicates of one another.

A better approach might be to have the test case identifier uniquely mapped to the path taken through the state machine model by the test generator. When the generator is run, if the path of a generated test case is

the same as the path of a test case already in the test management system, and only the test code is different, they would be considered the same test case and would be assigned the same test identifier, and the new test case would overwrite the old test case.

It would also be helpful to have a way to identify test cases already in the database that are no longer valid in the sense that they could not possibly be generated by the current model, because the path or the test code corresponding to the test case no longer exists. Such test cases could be flagged as invalid in some manner so that they are not executed on current or future versions of the software under test. In effect this would be using the test generator in reverse, to verify an existing set of test cases against the model rather than to generate new test cases from the model.

Applying such concepts would require including path information as well as test code in the output of a test generator, and storing test generator path information in the test management system. In general, it would require that test generation and test management tools be more tightly coupled than they are now.

7. Summary

Model-based testing was applied to testing of print driver installers in two case studies in a software testing organization. The software was modeled as an extended finite state machine and used to generate two test suites, one generated manually, and another generated using the TestMaster modeling and automated test generation tool. Both suites proved effective at finding defects in the software, but the TestMaster suite proved easier to maintain as the software changed.

The case studies did not immediately lead to wider use of model-based testing in the software test organization, in part because model-based testing tools require a unique set of skills, and because integration of test generation tools with test management systems still has unresolved issues in practice, especially in the area of test maintenance. While further work is needed in these areas, the potential of model-based testing was clearly demonstrated in the case studies.

About the Author
Chris Struble is a software design engineer with Hewlett Packard in Boise, Idaho, where he develops software test management and software test automation tools for the LaserJet™ Test Lab.

Acknowledgements

I wish to acknowledge the significant work done by Chris Bunsen to investigate TestMaster and point the way to how it could be used in printer software testing.
John Schroeder of Teradyne was a big help in getting me started with Testmaster. Tolga Gurel was a kind host during my trip to Teradyne for TestMaster training.
Thanks to Harry Robinson of Microsoft, master of http://www.model-based-testing.org, for encouraging me to write this paper.
Thanks to my manager Irv Tyrrell for supporting this investigation when the benefits were not proven. And special thanks to my wife LeAnne for putting up with too many late nights with her husband away.

References

[Clarke 1998] Clarke, James M., "Automated Test Generation from a Behavioral Model", Software Quality Week Conference, May 1998.
[El-Far and Whittaker 2001] Ibrahim K. El-Far and James A. Whittaker, "Model-based Software Testing" Quality Week Europe, Nov 2001. Available online at http://testingresearch.com/Ibrahim/papers_html/encyclopedia.htm
[Gurel 1999] Gurel, Tolga, "Fully Automated Requirements Tracing from TestMaster to TestExpert." Teradyne Users' Group Conference, May 1999.
[Mosley, 1997] Dan Mosley, Test -Rx Standardized Software Testing Process Methodology, CSST Technologies, online at http://www.csst-technologies.com/genericTest_Suite_Maintenance.html
[Sipser, 1997] Michael Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, 1997