Applying Models in your Testing Process

Steven Rosaria
Harry Robinson
Intelligent Search Test Group
Microsoft Corporation
srosaria@microsoft.com
harryr@microsoft.com

Abstract

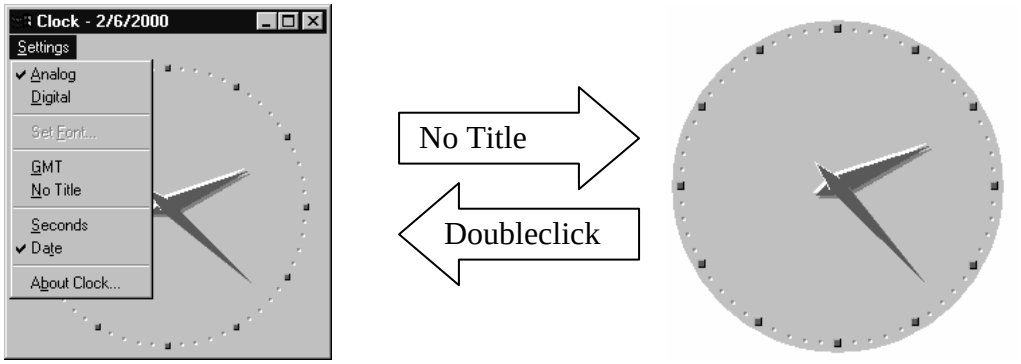
Model-based testing allows large numbers of test cases to be generated from a description of the behavior of the system under test. Given the same description and test runner, many types of scenarios can be exercised and large areas of the application under test can be covered, thus leading to a more effective and more efficient testing process.

The Current State of Software Testing

The following techniques are common in conventional software testing:

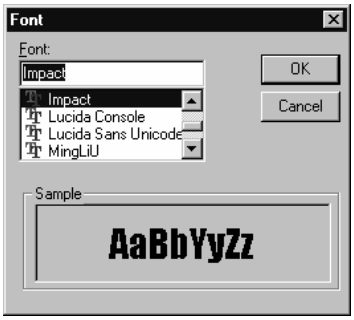
- **Handcrafted static test automation**: the same fixed set of test cases is executed on the system under test.
- **Random input generation**: test runners that can simulate keystrokes and mouse clicks bombard the application under test with random strings and click on arbitrary spots on the screen. This category also includes test runners that call API functions in random order with random parameters. The test runners simply apply one input after the other; they don't know what happens after an input is applied.
- **Hands-on testing**: an army of ad hoc testers executes test cases interactively against the system under test.

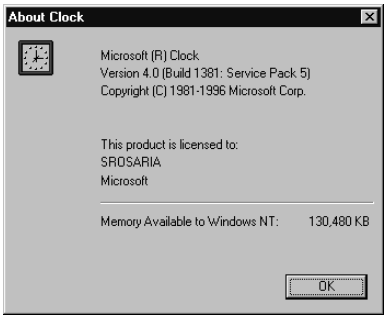
Let's take a look at how these techniques might be applied to the Microsoft Windows® clock application. The clock comes with a menu that allows the user to toggle a second hand, date, and GMT time display. The display of the clock can be changed from analog to digital and vice versa; the clock can be displayed without a title bar, where the user can double click to go back to the full display:



There is an option to set the display font, which brings up a dialog box for this purpose. This can only be done when the clock is in digital setting.

Finally there is a simple “about” box that can be closed.





Static test automation for the clock could be in the form of a script that simply tries the same sequence of actions in the exact same order each time. One such sequence might look like this:

- Start the clock
- Select Analog from the menu
- Select Digital from the menu
- Bring up the Font dialog box
- Select the third font in the list box
- Click OK in the Font dialog box
- Close the clock

Each script is entirely fixed and has to be maintained individually; there is absolutely no variation in the sequence of inputs.

One possible implementation of a random test runner is a test runner that simulates pressing the combination of ALT and all letters of the alphabet in turn in order to try and cover all menu bar options of an application under test. Many of those combinations may be undefined. Also, if a menu option brings up a new window that does not have a menu

of its own, the test runner is essentially wasting time trying to find menu choices. In a similar fashion, the test automation may try to enter a text string in an input control that is expecting floating-point numbers. While these are valid tests in and of themselves, it would be nice to be able to control the inputs being applied. In other words, the test runners have no idea of what is physically possible in the software under test, or what to expect after executing an action; they just apply the input and keep going. Random test automation might for example produce the following action sequence:

- Start the clock
- Type the string “qwertyuiop” (even though there isn’t a text box anywhere to enter data)
- Press ALT-A, ALT-B, ALT-C, etc; nothing happens, the menu is activated only by ALT-S
- After the menu is activated, press F, which brings up the Font dialog box
- Press ALT-A, ALT-B, ALT-C, etc; nothing happens, these shortcut keys are not defined in the Font dialog box
- Click on random spots of the screen

If this goes on long enough, eventually the test runner gets the clock application back in the main window and the whole process keeps going on and on until it is stopped by uncovering a bug, by some form of timeout, or by a command from the tester.

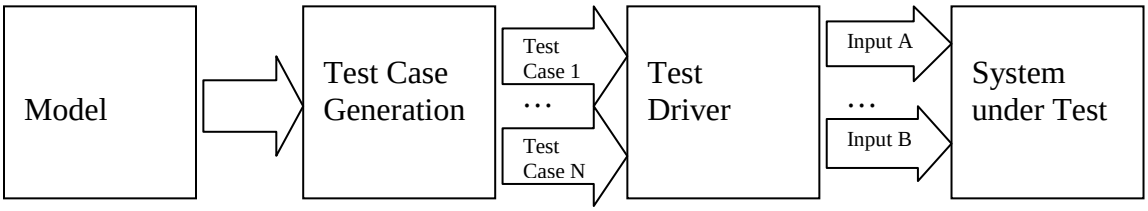
Hands-on execution of test cases consists of going through all features of the clock application and verifying the correctness by visually comparing the actual behavior with the expected behavior. For a simple application such as the clock, this approach actually gives reasonable coverage at a very low cost.

Test methods such as static test automation, random input generation, and hands-on testing have some major drawbacks:

- The system under test is constantly changing in functionality. This means that existing handcrafted static test automation must be adapted to suit the new behavior, which could be a costly process.
- Handcrafted static test automation implements a fixed number of test cases that can detect only specific categories of bugs, but the tests become less useful as these bugs are found and fixed. This phenomenon is referred to as the “pesticide paradox” [1]. A number of interesting bugs have been found in the clock application simply by varying the input sequence, as described in [2].
- Applying inputs at random makes it difficult to control input sequencing in an organized manner, which may lead to decreased test coverage. The entire sequence of random choices is indeed controlled by a seed value, but it is a process of trial and error to find a seed that ultimately results in a test sequence that consists entirely of actions that are valid for the system under test.
- Hands-on test execution does not scale well to complex systems.

Model-based testing solves these problems by providing a description of the behavior, or model, of the system under test. A separate component then uses the model to generate

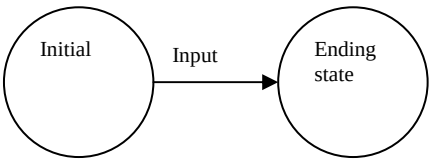
test cases. Finally, the test cases are passed to a test driver or test harness, which is essentially a module that can apply the test cases to the system under test.



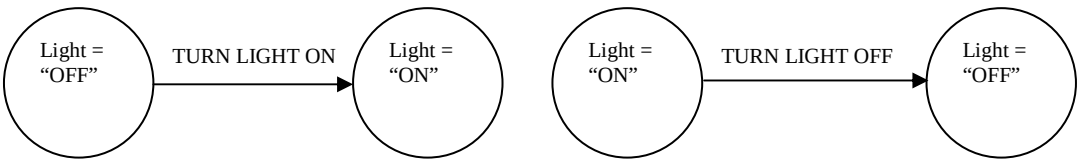
Given the same model and test driver, large numbers of test cases can be generated for various areas of testing focus, such as stress testing, regression testing, and verifying that a version of the software has basic functionality. It is also possible to find the most time-efficient test case that provides maximum coverage of the model. As an added benefit, when the behavior of the system under test changes, the model can easily be updated and it is once again possible to generate entire new sets of valid test cases.

An Introduction to Model-Based Testing

The main premise behind model-based testing is to create a model, a representation of the behavior of the system under test. One way to describe the system behavior is through variables called operational modes [3][4]. Operational modes dictate when certain inputs are applicable, and how the system reacts when inputs are applied under different circumstances. This information is encapsulated in the model, which is represented by a finite state transition table. In this context, a *state* is a valid combination of values of operational modes. Invalid combinations of these values represent situations that are physically impossible for the system under test, and are therefore excluded. Each entry in the state transition table consists of an initial state, an input that is executed, and an ending state that describes the condition of the system after the input is applied:



The model is created by taking any valid combination of values of operational modes as the initial state. All inputs that are applicable from this state are then listed, along with the new state of the software after the input is applied. As an analogy, think of a light switch that can be turned on or off. When the light is on, it can't be turned on again; similarly when the light is already off, it can't be turned off:



By repeating this process is for all states, a state transition table is formed that describes in great detail how the system under test behaves under all circumstances that are physically possible within the confines of the areas being modeled.

Model-based testing is very nimble and allows for rapid adaptation to changes to the system under development. As the software under test evolves, static tests would have to be modified whenever there is a change in functionality. With model-based testing, behavioral changes are handled simply by updating the model. This applies especially well to temporary changes in the system under test. For example, if a certain area of the system under test is known to be broken on a given build, static tests would keep running into the same problem, or those particular commands to the test runner would have to be rewritten to keep that from happening. On the other hand, if testing is done using a model, inputs that lead to the known faulty areas can be temporarily disabled from the model. Any test cases based on this new model will avoid the known errors and not a single line of test code has to be changed.

Model-based testing is resistant to the pesticide paradox, where tests become less efficient over time because the bugs they were able to detect have been fixed. With model-based testing, test cases are generated dynamically from the model. Each series of tests can be generated according to certain criteria, as explained later in this paper. In addition, model-based testing can be useful to detect bugs that are sensitive to particular input sequences. It is very important to note the fact that these additional benefits come at no extra cost to the model or the test harness. This means that once the initial investment has been made to create a model and a test harness, they are modified only sporadically. Meanwhile, the same model can generate large numbers of test cases, which can then be applied using the same test harness.

Developing a model is an incremental process that requires the people creating the model to take into consideration many aspects of the system under test simultaneously. A lot of behavioral information can be reused in future testing, even when the specifications change. Furthermore, modeling can begin early in the development cycle. This may lead to discovering inconsistencies in the specification, thus leading to correct code from the very outset.

To a certain extent, the technique of applying inputs randomly offers an alternative to static tests [5]. However, these types of test automation are not aware of the state of the system under test; for example, the test automation does not know what window currently has the focus and what inputs are possible in this window. Because of this, they will often try to do things that are illegal, or they exercise the software in ways it will never be used. In other words, it’s difficult to guide random input test automation in a cost-efficient way precisely because of its purely random nature. Another consequence of this unawareness of state is that random input test automation can only detect crashes, since it doesn’t know how the system works. The test automation is only able to apply inputs, but it does not know what to expect once an input has been applied. For example, the random test runner may execute a sequence of inputs in one particular window that brings up a new window that has a completely different set of possible inputs. Nevertheless, the test

runner is unaware of this fact, and keeps on applying random keystrokes and mouse clicks as if nothing had happened. On the other hand, model-based tests know exactly what is supposed to be possible at any point in time, because the model describes the entire behavior of the system under test. This in turn makes it possible to implement oracles of any level of sophistication. It is feasible to build a certain level of intelligence into the random test automation such that it will only try to apply inputs that are physically possible. The test automation can keep track of the window that currently has input focus and constrain the inputs that it will try to execute based on this knowledge. The disadvantage of the random input generation method is that when the system under test changes in behavior, the test automation has to be modified accordingly.

The disadvantages of model-based testing are that it requires that testers be able to program, and that a certain amount of effort is needed to develop the model for all but the simplest software systems; however, there are simple ways to minimize this effort [2].

Applying Model-Based Testing Principles to Software: an Example

The process of developing model-based software test automation consists of the following steps:

1. Exploring the system under test
2. Domain definition: enumerating the system inputs
3. Developing the model
4. Generating test cases by traversing the model
5. Execution and evaluation of the test cases

This entire process of developing a model will now be explained using the Microsoft Windows® clock application.

Exploring the System Under Test

In this stage the people creating the model get a general impression of the system's functionality, either by using the application or by going through the specification. For each input they encounter, they make comments that answer two questions that are key in describing the system behavior:

- When is the input available?
- What happens after the input is applied?

Let's take a look at the behavior of the clock starting from the basics. After launching the clock application, a user is in the "main" clock window, which displays the time of day. The size of the clock window will be the same as it was the last time the clock application was closed. The same rule also applies to the availability of a title bar. All the menu inputs can be applied any time if the clock is in the main window and a title bar is

available. The only exception to this rule is the option to set the font, which requires that the clock be in digital mode. This input and the "About Clock…" input are the only actions that bring up a new window.

The size of the clock window can be minimized, maximized, or restored. When the clock is minimized and then restored, the window will go back to whichever size it was before being minimized.

The clock can appear without a title bar by selecting "No Title" from the menu or by double clicking on the clock face. If no title bar is available, the only things a user can do are to double click to go back to full display or exit the clock by pressing ALT-F4.

Domain Definition

Following is an enumeration of the interesting system inputs:

Input Name	**Description**
Analog	Change to analog display
Digital	Change to digital display
Set_Font	Set the display font. Available only for digital setting
GMT	Display the time in Greenwich Mean Time format.
No_Title	Display without title bar. When the clock does not have a title bar, the only inputs possible are double click to go back to full display, and exit by pressing Alt-F4.
Seconds	Toggle the seconds display
Date	Toggle the date display
About	Bring up the About box
DoubleClick	Toggle between title bar and clock-only display
Font_OK	Click OK in the Font dialog box
Font_TypeFont	Type a random font name
Font_Cancel	Click Cancel in the Font dialog box
Font_SelectFont	Select a font at random from the font list box
About_OK	Click OK in the About box
Invoke	Launch the application. Applicable only when the clock is not running.
Terminate_Close	Exit the application by clicking the close window button
Terminate_Keystroke	Exit the application by pressing Alt-F4
Maximize	Maximize the application window. Can only be done if the window is not maximized already.
Minimize	Minimize the application window. Can only be done if the window is not minimized already.
Restore_Window	Restore the application window. Restores the window to its original size, which could be either the standard window size

	or the maximized state, depending on the previous window size. If the window starts up minimized and a Restore_Window input is applied, the clock always goes to the standard window size.

Developing the Model

The majority of inputs can of course only be applied only if the clock is running. Conversely, the only action that can be performed when the clock is not running is to start it. This can be encapsulated in the operational mode System = {Not_Invoked, Invoked}.

A different set of actions can be executed depending on which window currently has the focus, hence the operational mode Window = {Main, Font, About}.

The font can only be changed if the clock is displaying the time in digital format. This warrants the operational mode Setting = {Analog, Digital}.

The menu operations such as switching from analog to digital display, showing the second hand, and displaying the date are available only if there is a title bar, which leads to the operational mode Display = {All, Clock_Only}.

Finally, the size of the clock window can be maximized, minimized, or restored (brought back to its previous or normal size). If the clock window size is normal and it is minimized, the Restore_Window input brings the window size back to normal. On the other hand, if the clock is maximized and then minimized, the Restore_Window input brings the window back to the maximized size. This justifies the operational mode WindowSize = {Restored, Maximized, Minimized_From_Maximized, Minimized_From_Restored}

The operational modes and input set can be combined to form the following state transition table. Even though the state transition table looks complicated, [2] describes a technique that makes it easy to generate the state transition table.

Current State	**Input**	**Next State**
Not_Invoked Main Analog All Restored	Invoke	Invoked Main Analog All Restored
Not_Invoked Main Analog All Maximized	Invoke	Invoked Main Analog All Maximized
Not_Invoked Main Analog All Minimized_From_Maximized	Invoke	Invoked Main Analog All Minimized_From_Restored
Not_Invoked Main Analog All Minimized_From_Restored	Invoke	Invoked Main Analog All Minimized_From_Restored
Not_Invoked Main Analog Clock_Only Restored	Invoke	Invoked Main Analog Clock_Only Restored
Not_Invoked Main Analog Clock_Only Maximized	Invoke	Invoked Main Analog Clock_Only Maximized
Not_Invoked Main Digital All Restored	Invoke	Invoked Main Digital All Restored
Not_Invoked Main Digital All Maximized	Invoke	Invoked Main Digital All Maximized
Not_Invoked Main Digital All	Invoke	Invoked Main Digital All Minimized_From_Restored

Minimized_From_Maximized		
Not_Invoked Main Digital All Minimized_From_Restored	Invoke	Invoked Main Digital All Minimized_From_Restored
Not_Invoked Main Digital Clock_Only Restored	Invoke	Invoked Main Digital Clock_Only Restored
Not_Invoked Main Digital Clock_Only Maximized	Invoke	Invoked Main Digital Clock_Only Maximized
Invoked Main Analog All Restored	Analog	Invoked Main Analog All Restored
Invoked Main Analog All Restored	Digital	Invoked Main Digital All Restored
Invoked Main Analog All Restored	GMT	Invoked Main Analog All Restored
Invoked Main Analog All Restored	No_Title	Invoked Main Analog Clock_Only Restored
Invoked Main Analog All Restored	Seconds	Invoked Main Analog All Restored
Invoked Main Analog All Restored	Date	Invoked Main Analog All Restored
Invoked Main Analog All Restored	About	Invoked About Analog All Restored
Invoked Main Analog All Restored	DoubleClick	Invoked Main Analog Clock_Only Restored
Invoked Main Analog All Restored	Terminate_Close	Not_Invoked Main Analog All Restored
Invoked Main Analog All Restored	Terminate_Keystroke	Not_Invoked Main Analog All Restored
Invoked Main Analog All Restored	Maximize	Invoked Main Analog All Maximized
Invoked Main Analog All Restored	Minimize	Invoked Main Analog All Minimized_From_Restored
Invoked Main Analog All Maximized	Analog	Invoked Main Analog All Maximized
Invoked Main Analog All Maximized	Digital	Invoked Main Digital All Maximized
Invoked Main Analog All Maximized	GMT	Invoked Main Analog All Maximized
Invoked Main Analog All Maximized	No_Title	Invoked Main Analog Clock_Only Maximized
Invoked Main Analog All Maximized	Seconds	Invoked Main Analog All Maximized
Invoked Main Analog All Maximized	Date	Invoked Main Analog All Maximized
Invoked Main Analog All Maximized	About	Invoked About Analog All Maximized
Invoked Main Analog All Maximized	DoubleClick	Invoked Main Analog Clock_Only Maximized
Invoked Main Analog All Maximized	Terminate_Close	Not_Invoked Main Analog All Maximized
Invoked Main Analog All Maximized	Terminate_Keystroke	Not_Invoked Main Analog All Maximized
Invoked Main Analog All Maximized	Minimize	Invoked Main Analog All Minimized_From_Maximized
Invoked Main Analog All Maximized	Restore_Window	Invoked Main Analog All Restored
Invoked Main Analog All Minimized_From_Maximized	Terminate_Keystroke	Not_Invoked Main Analog All Minimized_From_Maximized
Invoked Main Analog All Minimized_From_Maximized	Maximize	Invoked Main Analog All Maximized
Invoked Main Analog All Minimized_From_Maximized	Restore_Window	Invoked Main Analog All Maximized
Invoked Main Analog All Minimized_From_Restored	Terminate_Keystroke	Not_Invoked Main Analog All Minimized_From_Restored
Invoked Main Analog All Minimized_From_Restored	Maximize	Invoked Main Analog All Maximized
Invoked Main Analog All Minimized_From_Restored	Restore_Window	Invoked Main Analog All Restored
Invoked Main Analog Clock_Only Restored	DoubleClick	Invoked Main Analog All Restored
Invoked Main Analog Clock_Only Restored	Terminate_Keystroke	Not_Invoked Main Analog Clock_Only Restored
Invoked Main Analog Clock_Only Maximized	DoubleClick	Invoked Main Analog All Maximized
Invoked Main Analog Clock_Only Maximized	Terminate_Keystroke	Not_Invoked Main Analog Clock_Only Maximized
Invoked Main Digital All Restored	Analog	Invoked Main Analog All Restored
Invoked Main Digital All Restored	Digital	Invoked Main Digital All Restored
Invoked Main Digital All Restored	Set_Font	Invoked Font Digital All Restored
Invoked Main Digital All Restored	GMT	Invoked Main Digital All Restored
Invoked Main Digital All Restored	No_Title	Invoked Main Digital Clock_Only Restored
Invoked Main Digital All Restored	Seconds	Invoked Main Digital All Restored
Invoked Main Digital All Restored	Date	Invoked Main Digital All Restored
Invoked Main Digital All Restored	About	Invoked About Digital All Restored
Invoked Main Digital All Restored	DoubleClick	Invoked Main Digital Clock_Only Restored
Invoked Main Digital All Restored	Terminate_Close	Not_Invoked Main Digital All Restored
Invoked Main Digital All Restored	Terminate_Keystroke	Not_Invoked Main Digital All Restored
Invoked Main Digital All Restored	Maximize	Invoked Main Digital All Maximized
Invoked Main Digital All Restored	Minimize	Invoked Main Digital All Minimized_From_Restored
Invoked Main Digital All Maximized	Analog	Invoked Main Analog All Maximized
Invoked Main Digital All Maximized	Digital	Invoked Main Digital All Maximized
Invoked Main Digital All Maximized	Set_Font	Invoked Font Digital All Maximized
Invoked Main Digital All Maximized	GMT	Invoked Main Digital All Maximized
Invoked Main Digital All Maximized	No_Title	Invoked Main Digital Clock_Only Maximized
Invoked Main Digital All Maximized	Seconds	Invoked Main Digital All Maximized
Invoked Main Digital All Maximized	Date	Invoked Main Digital All Maximized

Invoked Main Digital All Maximized	About	Invoked About Digital All Maximized
Invoked Main Digital All Maximized	DoubleClick	Invoked Main Digital Clock_Only Maximized
Invoked Main Digital All Maximized	Terminate_Close	Not_Invoked Main Digital All Maximized
Invoked Main Digital All Maximized	Terminate_Keystroke	Not_Invoked Main Digital All Maximized
Invoked Main Digital All Maximized	Minimize	Invoked Main Digital All Minimized_From_Maximized
Invoked Main Digital All Maximized	Restore_Window	Invoked Main Digital All Restored
Invoked Main Digital All Minimized_From_Maximized	Terminate_Keystroke	Not_Invoked Main Digital All Minimized_From_Maximized
Invoked Main Digital All Minimized_From_Maximized	Maximize	Invoked Main Digital All Maximized
Invoked Main Digital All Minimized_From_Maximized	Restore_Window	Invoked Main Digital All Maximized
Invoked Main Digital All Minimized_From_Restored	Terminate_Keystroke	Not_Invoked Main Digital All Minimized_From_Restored
Invoked Main Digital All Minimized_From_Restored	Maximize	Invoked Main Digital All Maximized
Invoked Main Digital All Minimized_From_Restored	Restore_Window	Invoked Main Digital All Restored
Invoked Main Digital Clock_Only Restored	DoubleClick	Invoked Main Digital All Restored
Invoked Main Digital Clock_Only Restored	Terminate_Keystroke	Not_Invoked Main Digital Clock_Only Restored
Invoked Main Digital Clock_Only Maximized	DoubleClick	Invoked Main Digital All Maximized
Invoked Main Digital Clock_Only Maximized	Terminate_Keystroke	Not_Invoked Main Digital Clock_Only Maximized
Invoked Font Digital All Restored	Font_OK	Invoked Main Digital All Restored
Invoked Font Digital All Restored	Font_TypeFont	Invoked Font Digital All Restored
Invoked Font Digital All Restored	Font_Cancel	Invoked Main Digital All Restored
Invoked Font Digital All Restored	Font_SelectFont	Invoked Font Digital All Restored
Invoked Font Digital All Maximized	Font_OK	Invoked Main Digital All Maximized
Invoked Font Digital All Maximized	Font_TypeFont	Invoked Font Digital All Maximized
Invoked Font Digital All Maximized	Font_Cancel	Invoked Main Digital All Maximized
Invoked Font Digital All Maximized	Font_SelectFont	Invoked Font Digital All Maximized
Invoked About Analog All Restored	About_OK	Invoked Main Analog All Restored
Invoked About Analog All Maximized	About_OK	Invoked Main Analog All Maximized
Invoked About Digital All Restored	About_OK	Invoked Main Digital All Restored
Invoked About Digital All Maximized	About_OK	Invoked Main Digital All Maximized

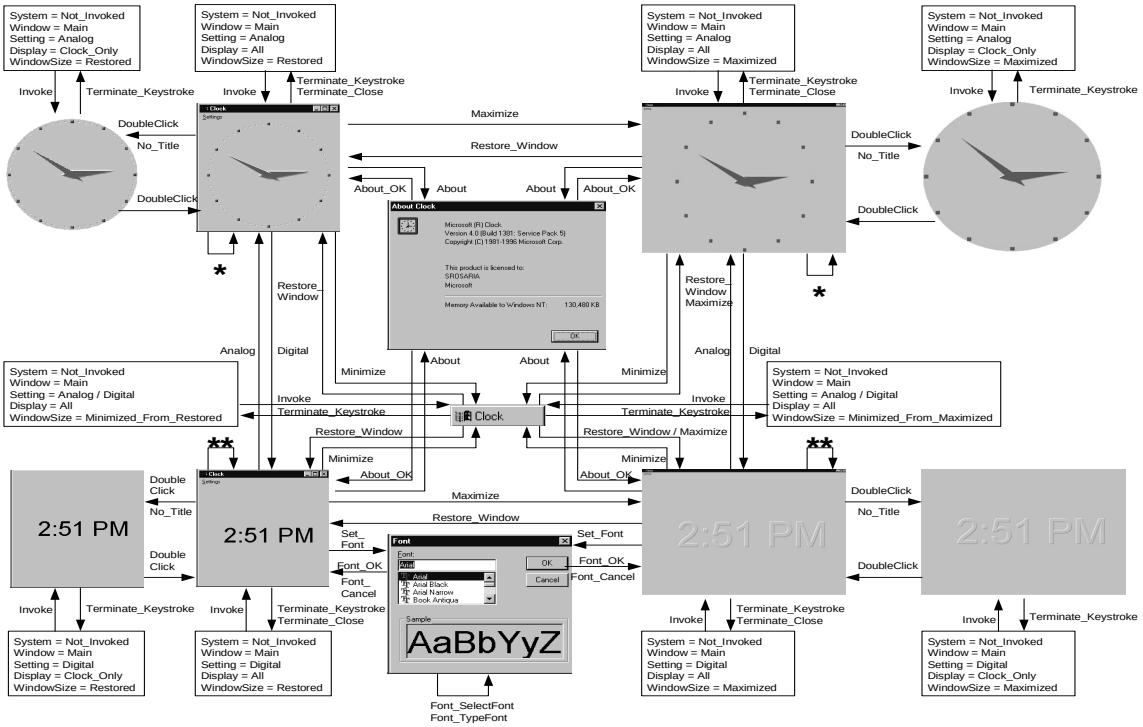
Here is a simplified diagram of the state transition table. The About window and the minimized clock state in the diagram actually represent 4 states each, but since they look the same to a user, they have been condensed into one screen shot. Similarly, the Font window represents 2 states; finally, the Not_Invoked states on either side of the minimized clock state apply to both the Analog and Digital values for Setting.

The arcs labeled * represent the inputs:

Analog
GMT
Seconds
Date

The arcs labeled ** represent the inputs:

Digital
GMT
Seconds
Date



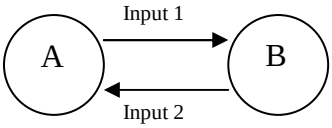
Generating Test Cases by Traversing the Model

Once a model is sufficiently developed to be useful, the same model can be used to generate large numbers of test cases. Essentially the model can be considered a graph, and a variety of graph traversal algorithms can be used to navigate the model and produce an input sequence, or test case. Here are some examples:

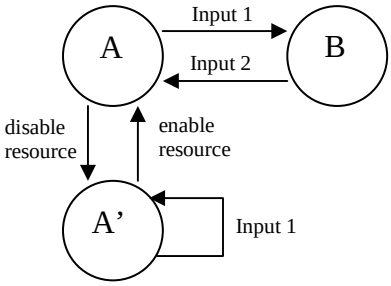
- The *Chinese Postman* algorithm is the most efficient way to traverse each link in the model. Speaking from a testing point of view, this will be the shortest test sequence that will provide complete coverage of the entire model [6]. An interesting variation is called the *State-changing Chinese Postman* algorithm, which looks only for those links that lead to different states (i.e. it ignores self-loops).
- The *Capacitated Chinese Postman* algorithm can be used to distribute lengthy test sequences evenly across machines [7].
- The *Shortest Path First* algorithm starts from the initial state and incrementally looks for all paths of length 2, 3, 4, etc. This is essentially a depth-first search.
- The *Most Likely Paths First* algorithm treats the graph as a Markov chain. All links are assigned probabilities and the paths with higher probabilities will be executed first. This enables the automation to be directed to certain areas of interest [6].

Extending the Model

Other areas of testing focus can also be implemented by incorporating their requirements in the model. It is possible to do stress testing with models by creating abstract modes that represent the availability of a certain resource. One could have a special-purpose component that once activated, simulates failure for each request of that particular resource. Given a simplistic system with 2 states, A and B, and 2 inputs:



A special input, let's call it "disable resource", is then included in the model, which will activate the custom tool that will simulate failures. Conversely, there is also an input "enable resource" that does the opposite. The "disable_resource" input will take the system under test into a resource-restricted state where it will be possible to examine how the system deals with failure on resource requests:



The resource in question could be for example the availability of a communications line or system memory. By adding the inputs controlling the availability of the resource to the model, a graph traversal algorithm will then include them in the test case it generates. The state A' represents the system operating in this resource-constrained environment. Input 1 applied in state A' has a different outcome from the same input applied in state A. This approach exercises the exception handling of the system under test and gives an accurate measure for the robustness of the system. An argument can be made that the same type of stress testing can be achieved by disabling the resource from the very beginning and then start running tests. Under these circumstances, *all* inputs will be executed without that resource. By incorporating stress test situations inside a model, the robustness of *specific* inputs can be tested with pinpoint accuracy.

Abstracting Inputs

The same concept of creating abstract modes and special-purpose inputs can also be used to do boundary value testing with models. Suppose it is necessary to test an API function called XYZ that takes an integer number as argument. One can define abstract inputs, each of which handles an equivalence class of the input domain, namely the integer data type:

- XYZNegativeBoundary: calls the API XYZ with the smallest negative number as argument
- XYZZeroBoundary: calls the API XYZ with the argument 0

- XYZIntegerInRange: calls the API XYZ with an integer in the valid range
- XYZPositiveBoundary: calls the API XYZ with the largest positive number as argument

Extensive functionality testing, regression testing, and verifying that a version of the software has basic functionality can be covered by the Most Likely Paths First algorithm. The inputs that a user is most likely to apply, or inputs of vital importance are assigned high probabilities, so the graph traversal algorithm will select them more frequently. Assigning high probabilities only to certain inputs in effect provides a fixed, predetermined path that will be the same each time; note that this serves essentially the same purpose as static tests, namely to verify that a few basic scenarios work, thereby ensuring a minimum acceptable level of functionality. Also, tests can be focused on areas that are used very often by weighting those specific areas more heavily.

Special Uses of the Model
The model itself can be altered to support different test strategies. For instance, the input that terminates the system under test (let's call it the "exit" input) appears in the model. By removing all transitions of the exit input from the model, one can generate long input sequences that essentially become tests to determine the continuous hours of operation.

One can easily implement a random graph traversal to achieve random testing using a model. The advantage here is that no time will be wasted trying to apply unavailable inputs, like typing garbage text strings in an input box that is expecting a floating-point number. Model-based test automation knows at all times what is and what isn't possible, and what to expect after each action.
Finally, an algorithm can be implemented to take the shortest path to a specific part of the model and then roam around in that area. The test case produced by such an algorithm would make an excellent test case for retesting after a bug fix.

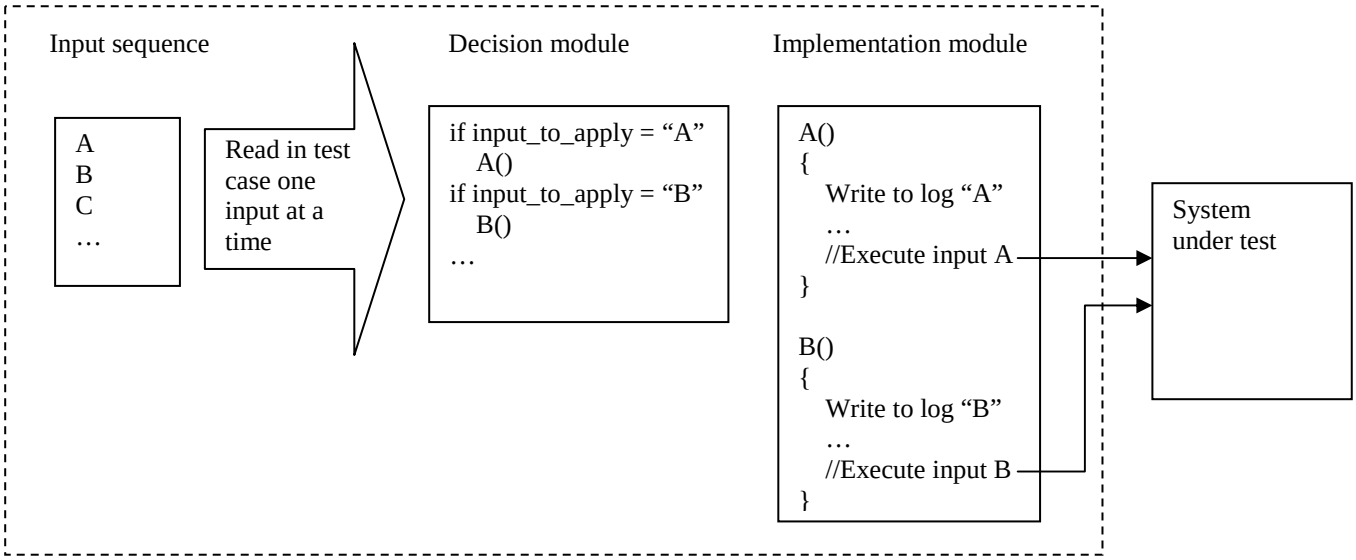
An interesting point is that particular graph traversal algorithms can be used to check the validity of the model itself; in other words, by traversing the model it is possible to verify whether all states are reachable from all other states.

It deserves mention once again that reliability, regression, verifying that a build has a minimum level of functionality, and all these other areas of testing focus and types of model coverage mentioned above come at no extra cost once a model and harness have been developed. The greatest effort at that point goes into actually running test cases and continued development of a suite of reusable graph traversal algorithms. An in-depth discussion on the topic of graph traversal techniques is presented in [6].

Execution and Evaluation of the Test Cases

This is the stage where the system is actually tested. Testing is done by a test harness or test driver, a program that can apply an input sequence to the system under test. A test

driver consists of a decision module and an implementation module. Here is a possible implementation of a test driver outlined by a dashed rectangle:



The decision module gets a test sequence from any one of the graph algorithms mentioned in the preceding section. It reads the test sequence input by input, determines which action is to be applied next, and calls the function in the implementation module that performs that input. The implementation module logs the action it is about to perform and then executes that input on the system under test. Next, it verifies whether the system under test reacted correctly to the input. Since the model accurately describes what is supposed to happen after an input is applied, oracles can be implemented at any level of sophistication.

A test harness designed in this particular way is able to deal with any input sequence because its decision logic is dynamic. In other words, rather than always executing the same actions in the same order each time, it decides at runtime what input to apply to the system under test. Moreover, reproducing a bug is simply a matter of feeding as input sequence the execution log of the test sequence that caused or revealed the failure.

The implementation module contains the code that actually applies inputs to the system under test and verifies the results after the input is applied. As an example, here is some Rational Visual Test® code for the clock test driver that brings up the the “About” window:

```
Sub About()
        Print #log, "About"    ‘First write the action that is to be performed to the log

        WMenuSelect("&Settings\A&bout Clock...")     ‘Now apply the input

        ‘Verify that you're in the About window and halt if the caption is not “About
        ‘Clock”
        Dim strCaption As String
        strCaption = GetText(WGetActWnd(0))
        If strCaption <> "About Clock" Then FAIL "Incorrect window"

End Sub
```

The caption of the active window must change to “About Clock” after the “About” input is applied. If this is not the case then something went wrong somewhere. The test driver will report failure and stop execution, and its log can be examined for a reproduction scenario.

Invalid inputs can be handled in a similar manner. Here the correctness verification would check that an error dialog box pops up or assert that an API does indeed fail when invalid parameters are passed to it.

Model-Based Testing and Software Reliability Metrics

Using models improves reliability and facilitates reliability measurements. First of all, there are the commonly used continuous hours of operation and mean time to failure metrics. One distinct advantage of model-based testing shows up when measuring coverage. It is not possible to execute all combinations of test paths. However, it is feasible to measure what part of the model has been covered when generating test cases with graph traversal algorithms. At this stage, determining coverage is basically reduced to a sampling problem. Given the number of paths covered, an extrapolation can be made as to what portion of the model has been covered.

Model-based testing has historically been used in industries such as telecommunications and avionics, which have a stringent software quality bar. Case studies in those industries [8][9] have shown a ten-fold productivity improvement in reaching that quality level. “At this level of test generation productivity improvement, one test engineer using [a model-based testing tool] can be as productive as ten test engineers using manual test generation” [10].

The simple model creation and execution techniques described in the current paper make this level of quality improvement available to all areas of the software industry.

Conclusion

Model-based testing is an efficient and adaptable method of testing software by creating a model describing the behavior of the system under test. Large numbers of test cases can be generated from this model using various graph traversal algorithms. A test harness then executes these test cases against the system under test. Many areas of testing focus can be implemented and different levels of model coverage can be achieved by using the same model and test harness.

References

1. B. Beizer, "Software Testing Techniques", 2nd Edition 1990.
2. H. Robinson, "Finite State Model-Based Testing on a Shoestring", Proceedings of the Software Testing Analysis and Review Conference, San Jose, CA, Nov. 1999.
3. J. Whittaker, "Stochastic Software Testing", Annals of Software Engineering, 4, August 1997.
4. I. El-Far, “Automated Construction of Software Behavior Models”, Masters Thesis, Florida Institute of Technology, 1999.
5. N. Nyman, “GUI Application Testing with Dumb Monkeys”, Proceedings of STAR West 1998.
6. H. Robinson, "Graph Theory Techniques in Model-Based Testing", 1999 International Conference on Testing Computer Software.
7. D. Dill, R. Ho, M. Horowitz, and C. Yang, “Architecture Validation for Processors”, Proceedings of the 22nd annual International Symposium on Computer Architecture, 1995.
8. L. Apfelbaum, and J. Doyle, “Model-Based Testing”, Presented at Software Quality Week, 1997.
9. P. Savage, S. Walters, and M. Stephenson, "Automated Test Methodology for Operational Flight Programs", Presented at IEEE Aerospace Conference, 1997.
10. J. Clarke, “Automated Test Generation from a Behavioral Model”, Presented at Software Quality Week, 1998.