

Tools & Automation

Intelligent Test Automation

A model-based method for generating tests from a description of an application's behavior *by Harry Robinson*

Warning: The fairy tale you are about to read is a fib—but it's short, and the moral is true.

Once upon a product cycle, there were four testers who set out on a quest to test software.



Tester 1

started hands-on testing immediately, and found some nice bugs. The development team happily fixed these bugs, and gave Tester 1 a fresh version of the software to test. More testing, more bugs, more fixes.

Tester 1 felt productive, and was happy—at least for a while.

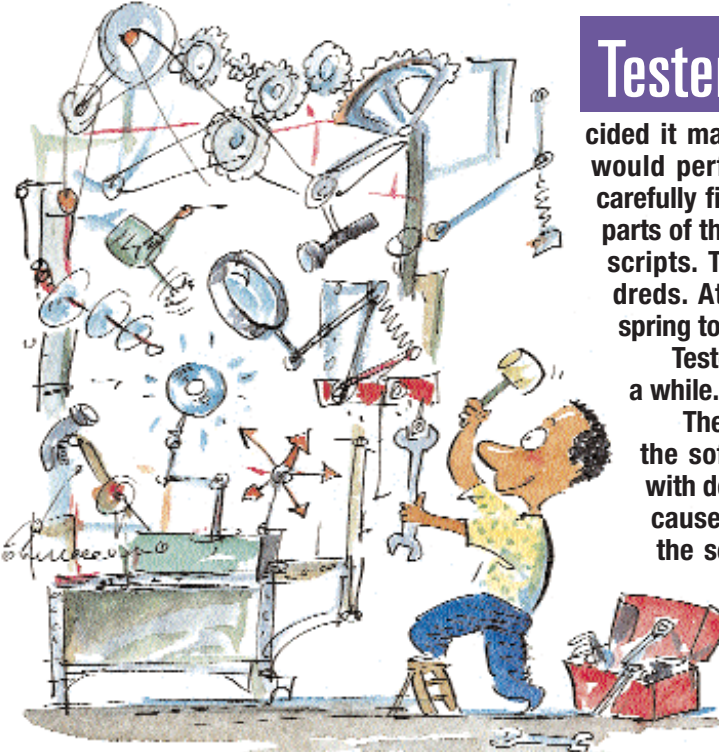
After several rounds of this find-and-fix cycle, he became bored and bleary-eyed from running virtually the same tests over and over again by hand. When Tester 1 finally ran out of enthusiasm—and then out of patience—the software was declared “ready to ship.”

Customers found it too buggy and bought the competitor's product.

ILLUSTRATIONS BY STEVE BJÖRKMAN

►► QUICK LOOK

- Improving the efficiency of your automated testing through modeling
- Overcoming the limitations of hands-on and static automation testing

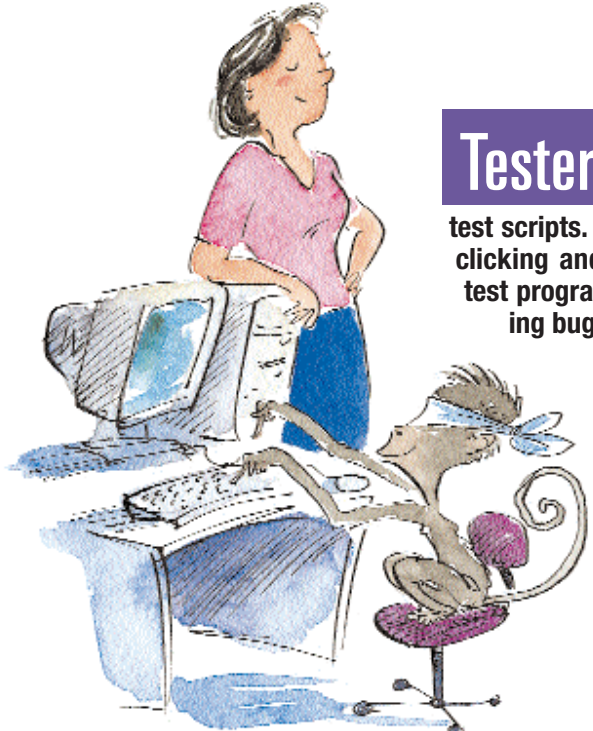


Tester 2

started testing by hand, but soon decided it made more sense to create test scripts that would perform the keystrokes automatically. After carefully figuring out tests that would exercise useful parts of the software, Tester 2 recorded the actions in scripts. These scripts soon numbered in the hundreds. At the push of a button, the scripts would spring to life and run the software through its paces. Tester 2 felt clever, and was happy—at least for a while.

The scripts required a lot of maintenance when the software changed. He spent weeks arguing with developers to stop changing the software because it broke the automated tests. Eventually, the scripts required so much maintenance that there was little time left to do testing.

When the software was released, customers found lots of bugs that the scripts didn't cover. They stopped buying the product and decided to wait for version 2.0.



Tester 3

didn't want to maintain hundreds of automated test scripts. She wrote a test program that went around randomly clicking and pushing buttons in the application. This "random" test program was hypnotic to watch, and it found a lot of crashing bugs.

Tester 3 enjoyed uncovering such dramatic defects, and was happy—at least for a while.

Since the random test program could only find bugs that crashed the application, Tester 3 still had to do a lot of hands-on testing, getting bored and bleary-eyed in the process. Customers found so many functional bugs in the software when it was released that they lost trust in the company and stopped buying its software.

Tester 4

began with hands-on, exploratory testing to become familiar with the application—and used the knowledge gained during the hands-on testing to create a very simple behavioral model of the application. Tester 4 then used a test program to test the application's behavior against what the

model predicted. The behavioral model was much simpler than the application under test, so it was easy to create. Since the test program knew what the application was supposed to do, it could detect when the application was doing the wrong thing.

As the product cycle progressed, developers wrote new features for the application. Tester 4 quickly updated the model, and the tests continued running. The program ran day and night, constantly generating new test sequences. Tester 4 was able to run the tests on a dozen machines at once and get several days of testing done in a single night.

After several rounds of testing and bug fixes, Tester 4's test generator began to find fewer bugs. Tester 4 upgraded the model to test for additional behaviors and continued testing. Tester 4 also did some hands-on testing and static automation for those parts of the application which were not yet worth modeling.

When Tester 4's software was released, there were very few bugs to be found. The customers were happy. The stockholders were happy.

And Tester 4 was happy.



Commentaries

These four scenes show some of the approaches available in software testing today.

Tester 1 is a typical hands-on tester, manually running all tests from the keyboard. Hands-on testing is common throughout the industry today—it provides immediate benefits, but in the long run it is tedious for the tester and expensive for the company.

“One of the saddest sights to me has always been a human at a keyboard doing something by hand that could be automated. It’s sad but hilarious.”

—Boris Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*

Tester 2 practices what I call “static test automation.” Static automation scripts exercise the same sequence of com-

mands in the same order every time. These scripts are costly to maintain when the application changes. The tests are repeatable; but since they always perform the same commands, they rarely find new bugs.

“Highly repeatable testing can actually minimize the chance of discovering all the important problems, for the same reason that stepping in someone else’s footprints minimizes the chance of being blown up by a land mine.”

—James Bach, “Test Automation Snake Oil,” *Windows Tech Journal*, October 1996

Tester 3 operates closer to the cutting edge of automated testing. These types of “random” test programs are called *dumb monkeys* because they essentially bang on the keyboard aimlessly. They come up with unusual test action sequences and find many crashing bugs, but it’s hard to

direct them to the specific parts of the application you want tested. Since they don't know what they are doing, they miss obvious failures in the application.

"Monkey testing should not be your only testing. Monkeys don't understand your application, and in their ignorance they miss many bugs."

—Noel Nyman, "Using Monkey Test Tools," *STQE*, January/February 2000

Tester 4 combines the other testers' approaches with a type of intelligent test automation called "model-based testing."

Model-based testing doesn't record test sequences verbatim like static test automation does, nor does it bang away at the keyboard blindly. Model-based tests use a description of the application's behavior to determine what actions are possible and what outcome is expected. This automation generates new test sequences endlessly, adapts well to changes in the application, can be run on many machines at once, and can run day and night.

"[An artist] paints with his brain, not with his hands."
—Michelangelo Buonarroti

The Moral of the Story

Tester 1's method required that his hands always be at work on the keyboard. Eventually Tester 1's brain and hands gave out.

Tester 2's static scripts repeated keyboard actions that his hands had already performed.

Tester 3's monkeys were essentially brainless hands banging on the keyboard.

Tester 4, on the other hand, supplemented the others' techniques by:

- thinking about the application's behavior,
- describing that behavior to a test generator, and
- letting the test generator create and run test cases.

By generating tests from a description of the application's behavior, Tester 4 could perform tests that were not practical under the other test approaches.

The moral of the tale: Automate your *brain*, not just your *hands*.

Use Your Brain

Let's look at an example of creating and using a behavioral model to test a software application.

Hands-on testing is a good way to start the test automation process. I call this phase "exploratory modeling" because it combines *exploratory testing* with the discovery of a *model* that can later be used to generate tests. As you begin to understand the behavior of each action, you can cre-

Hands-on testing is a good way to start the test automation process. I call this phase "exploratory modeling" because it combines exploratory testing with the discovery of a model that can later be used to generate tests. As you begin to understand the behavior of each action, you can create rules that will help you model and test the application.

ate rules that will help you model and test the application.

This is the essence of model-based testing: To describe the behavior you expect in a way that can be used to generate tests. Ask yourself the following two questions for every action you are going to test:

1. When is this action possible?
2. What is the outcome when this action is executed?

For instance, suppose you have been asked to test the behavior of files in a Windows folder. In particular, you are going to test the **Create**, **Delete**, and **Invert Selection** actions.

Modeling the "Create" Action

- *When is **Create** possible?* This example is kept simple by limiting the number of files in the folder to **1 File**. Therefore **Create** is only possible in this model when there are **0 Files** in the folder.
- *What is the outcome when **Create** is executed?* When you **Create** a new file in a folder, the number of files in the folder increases by one. The newly created file is initially Selected, so it appears highlighted in the folder. In fact, the new file is the only Selected file in the folder, no matter how many were Selected before the **Create** action.

Modeling the "Delete" Action

- *When is **Delete** possible?* **Delete** is only possible in this model when there is at least **1 Selected File** in the folder.
- *What is the outcome when **Delete** is executed?* When you execute the **Delete** action, any Selected file disappears from the folder.

Modeling the "Invert Selection" Action

- *When is **Invert Selection** possible?* **Invert Selection** is always possible in this model, even when there are **0 Files** in the folder.
- *What is the outcome when **Invert Selection** is executed?* When you execute the **Invert Selection** action, all Selected files in the folder become Unselected, and all Unselected files become Selected. When there are **0 Files** in the folder, **Invert Selection** leaves the folder unchanged.

Creating a State Model

You can now construct what is called a “state model” of the system’s behavior, as shown in Figure 1. It incorporates all the behaviors described above. Note the way the **Invert Selection** action loops from the **0 Files** State back to the **0 Files** State. That models the way **Invert Selection** does nothing if there’s nothing to invert.

Very Pretty. So What?

Now that you understand how the application works, you could manually test these actions and verify whether the Windows folder behaves as you expect. However, because your understanding is being carried around inside your head, your results are limited by your time and your stamina.

On the other hand, if you could somehow communicate this state model directly from your brain to a computer, it could generate and execute tests on the system for you.

Fortunately, this model can be represented in a format known as a “state table” that the computer can read. Each row of the state table (see Table 1) shows the Ending State that will result when an action is applied to the application in the Starting State.

Use the Computer’s Brain, Too

Once we have put the state model into a state table that the computer can understand, what can the computer do for us? How can we exploit our information about the application’s behavior?

The computer can use the state table to generate sequences of tests to run against the application. As you will see in the following examples, these test sequences can be chosen for their novelty, their efficiency of testing, or their exhaustiveness. This test generation is a powerful way to apply your understanding—and this is what model-based testing is all about.

A Random Walk Through the State Model

One simple way to generate test actions is to randomly select any available action from the current state of the application. For example, if you are in the **0 Files**

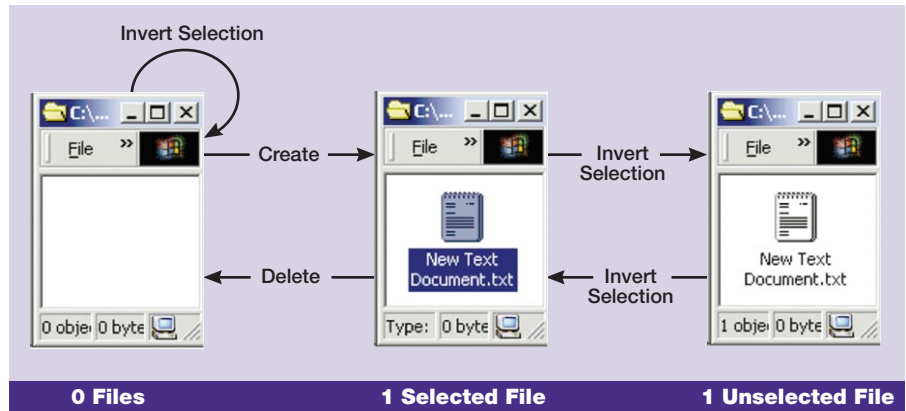


FIGURE 1 The state model

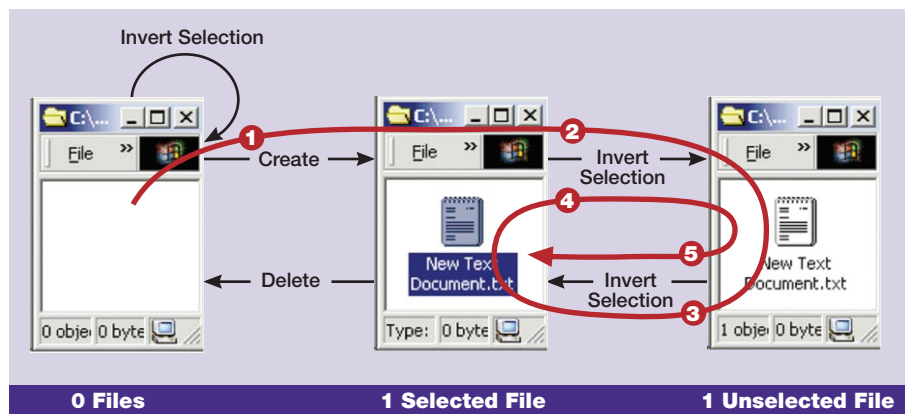


FIGURE 2 A random walk

Starting State, you can choose either of these two actions:

- **Invert Selection** (which leaves you in the **0 Files** State)
- **Create** (which leaves you in the **1 Selected File** State)

By choosing random actions in this way, you can generate many unusual sequences (just like Tester 3’s random “monkey test program”), and you will eventually exercise all of the actions in the model. Figure 2 shows a typical random walk. Notice that the random walk executed the same action (**Invert Selection**) four times in a row, but has

Starting State	Action	Ending State
0 Files	Invert Selection	0 Files
0 Files	Create	1 Selected File
1 Selected File	Invert Selection	1 Unselected File
1 Selected File	Delete	0 Files
1 Unselected File	Invert Selection	1 Selected File

TABLE 1 A state table for behavior of files in a Windows folder

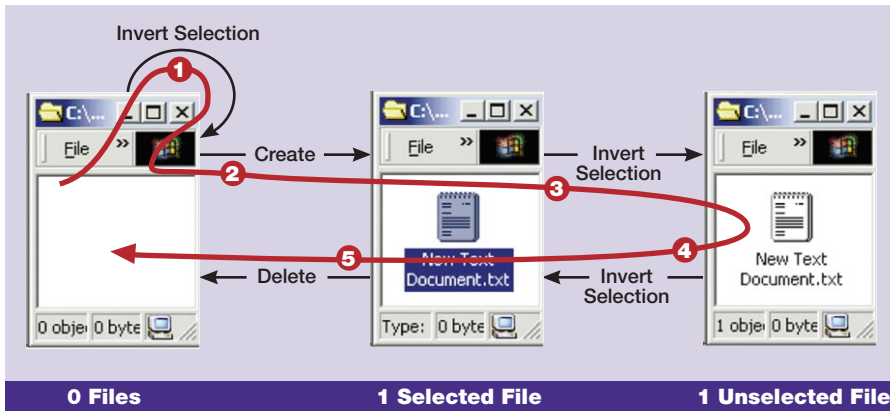


FIGURE 3 A Chinese Postman walk

so far left two other actions untouched. Such is the nature of random testing.

ACTION TO EXECUTE	ENDING STATE
1. Create	1 Selected File
2. Invert Selection	1 Unselected File
3. Invert Selection	1 Selected File
4. Invert Selection	1 Unselected File
5. Invert Selection	1 Selected File

An Efficient Walk Through the State Model: The Chinese Postman Walk

Random walks are inefficient at reaching all test actions when the model is large. How can we test each of the actions in the model efficiently?

This turns out to be the same problem a letter carrier faces when delivering mail. Imagine that each of the *actions* in the model is a street where mail must be delivered—and that each of the *states* in the model is an intersection where the letter carrier can change direction. Just as the letter carrier must travel down each street to deliver the mail, we must test each action in the model. And in both cases, we would like to minimize the amount of additional travel needed.

A Chinese mathematician named Kwan Mei-Ko formulated an elegant solution to this problem, and it is known as the Chinese Postman algorithm in his honor (see Figure 3). Kwan’s method generates a path through the state model that exercises every action in the model in the fewest number of steps. The test sequence listed below covers all five actions in the model in only five steps. This efficiency can be handy if you have a large application that you want to test quickly.

ACTION TO EXECUTE	ENDING STATE
1. Invert Selection	0 Files
2. Create	1 Selected File
3. Invert Selection	1 Unselected file
4. Invert Selection	1 Selected File
5. Delete	0 Files

An Even More Efficient Walk: The State-Changing Chinese Postman Walk

Some actions in a model—such as hitting **Invert Selection** with **0 Files** in the folder—do not change the state of the ap-

plication. If you think that bugs are more likely to occur where the application changes state, you may want to prioritize your efforts by first testing the state-changing actions.

A simple way to do this is to filter out from the state table any actions that don’t change the state. In Table 1, that would remove the first action (**Invert Selection**).

Running the Chinese Postman algorithm over this reduced state model generates a test sequence that covers all of the model’s state-changing actions in four steps—essentially removing the

first step of the previous example:

ACTION TO EXECUTE	ENDING STATE
1. Create	1 Selected File
2. Invert Selection	1 Unselected File
3. Invert Selection	1 Selected File
4. Delete	0 Files

Shortest Round Trips Back to the Starting State

Suppose you want to exhaustively test every sequence that takes the Windows folder from the **0 Files** State back to the **0 Files** State in a certain number of steps or less? Sequences like these that constantly generate new variations would be unthinkable for Tester 2’s static automation.

It is trivial for a computer to generate a list of such paths from the state model. You can generate sequences of increasing length as long as you have computer cycles to burn, probing deeper and deeper into the model.

Figure 4 shows all round trips that start at the **0 Files** State and have a path length less than or equal to four steps.

Path **A** has a length of one step:

A1: Invert Selection

Path **B** has a length of two steps:

B1: Create

B2: Delete

Path **C** has a length of four steps:

C1: Create

C2: Invert Selection

C3: Invert Selection

C4: Delete

Use the Computer’s Hands

The output of each of these algorithms is a sequence of test actions to execute. What would be the best way to perform these actions? You *could* hand a human tester the list of actions to execute by hand—but this would be slow, tedious, and cruel. Who would want to spend their day executing lists of actions? Such repetitious work is the kind of mind-numbing scenario that caused poor Tester 1 such grief.

ANNIE BISSETT

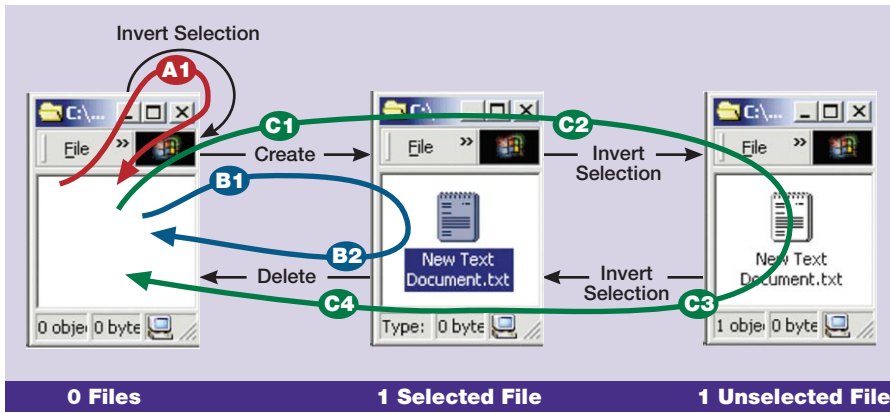


FIGURE 4 All round trips in four steps or less

Instead, you can write a simple test execution program that will read the list and then execute test code for each action in that list.

For instance, in Visual Test, the code to implement the **Create** action is:

```
WToolBarButtonClick("@1","File")    ' Open the File menu
WMenuItemSelect("New")              ' Select New File
WMenuItemSelect("Text Document")    ' Choose Text Document
Play "{Enter}"                      ' Accept the default filename
```

In typical static automation, this code would be embedded in a script—but in a model-based test program, this snippet of code is invoked whenever the list of test actions says to perform the **Create** action.

Use the Computer's Eyes

Automating the test actions is only half the battle. You also need an automated method of determining if the application is working correctly.

This method—a function that determines if the application has behaved correctly in response to a test action—is called a *test oracle*. Some test methods, such as Tester 3's random monkey test programs, must rely on crude test oracles such as whether the application has crashed.

Model-based testing gives the test program the ability to check for indicators of good behavior more subtle than “didn't crash.” From the information in the state table, the model “knows” what actions should be available from each state and the expected outcome of each action. For instance, the model says that the test program should be able to execute the **Delete** action from the **1 Selected File** State. The model also says that executing that **Delete** action should leave the application in the **0 Files** State. This knowledge provides two ways to verify that the application has behaved correctly.

First, the test program can detect if an action is not available when it should be. If the **Delete** action is not available when the ap-

plication is in the **1 Selected File** State, the test program will report an error because the test code will fail when it finds no menu selection for **Delete**.

Second, the model is always aware of what state the application should be in. Knowing the expected ending state of each action means that we can create test oracle routines to check (at the conclusion of each action) that the appropriate number of files are present and selected in the folder. For instance, when the **Delete** action above is executed, the Ending State should have **0 Files** in the folder (and of

course, **0 Files** Selected).

Programmatic test languages usually provide functions that allow the test program to verify aspects of the application. Two useful Visual Test functions for the current model are:

- **WViewCount()** which indicates the number of files in the folder, and
- **WViewItemSelected()** which tells how many files in the folder are Selected.

The test program can verify that the application is in the correct state, as shown in Table 2.

The **Delete** action discussed above should leave the application in the **0 Files** State. If **WViewCount()** returns a value other than 0, the test program oracle will report an error because the number of files in the folder is incorrect.

How to Update Model-based Tests

Remember how Tester 2's static test automation attempts were frustrated by application changes? Tester 4, in contrast, was able to adapt the model-based test automation quickly to changes in the application.

Incorporating New Actions into the Model

Suppose your development team tells you that they have implemented the **Select All** action. How should you update your tests for this new action? Simple—upgrade the state model to incorporate the **Select All** action, and regenerate the tests.

First, you model the **Select All** action by answering our two basic questions:

Application State	Expected Return Value of WViewCount()	Expected Return Value of WViewItemSelected()
0 Files	0	0
1 Selected File	1	1
1 Unselected File	1	0

TABLE 2 A state table showing Visual Test functions **WViewCount()** and **WViewItemSelected()**

1. When is **Select All** possible? **Select All** is always possible in this model, even when there are **0 Files** in the folder.
2. What is the outcome when **Select All** is executed? When you execute **Select All**, all the files in the folder become Selected. If there are **0 Files** in the folder, **Select All** leaves the folder unchanged. This is indicated in the illustration below, where the **Select All** action loops from the **0 Files** State back to the **0 Files** State.

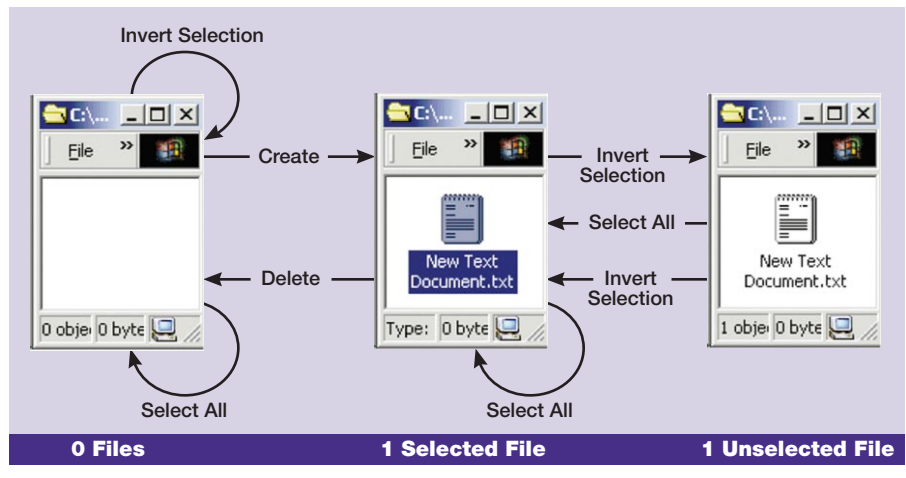


FIGURE 5 State model including **Select All**

Figure 5 shows the new state model with the **Select All** action incorporated.

Running the Chinese Postman algorithm on the updated model (see Figure 6) gives a nine-step test sequence—using the **0 Files** Starting State—that exercises every action in the model, including the new **Select All**:

ACTION TO EXECUTE	ENDING STATE
1. Invert Selection	0 Files
2. Create	1 Selected File
3. Invert Selection	1 Unselected File
4. Select All	1 Selected File
5. Invert Selection	1 Unselected File
6. Invert Selection	1 Selected File
7. Select All	1 Selected File
8. Delete	0 Files
9. Select All	0 Files

The next step would be to determine the code that is used to invoke the **Select All** action whenever it occurs in the test sequence. For Visual Test it would be as follows:

```
WToolBarButtonClick("@1","Edit")
WMenuSelect("Select All")
```

In Summary

It can take significant effort to understand and model an application. And it can be difficult to leave the easy path of hands-on testing or static automation long enough to invest time thinking about *how* to test that application—as we saw in the trials and tribulations of our fairy tale’s four testers.

The rewards, however, are great:

- Model-based testing creates flexible, useful test automation from practically the first day of development.

- Models are simple to modify, so model-based tests are economical to maintain over the life of a project.
- Models can generate innumerable test sequences tailored to your needs.
- Models allow you to get more testing accomplished in a shorter amount of time because a test generator can create and verify test sequences around the clock on multiple machines.
- Model-based testing can supplement other forms of testing, and can perform tests that aren't practical under other test approaches.

You and I know that software testing is no fairy tale, and that happily-ever-afters are never guaranteed. But adding model-based intelligence to your testing is a powerful tool to help you find your way toward your own happy ending. **STQE**

Harry Robinson is software test lead with the Intelligent Search Group at Microsoft. He maintains the Model-Based Testing Home Page (www.model-based-testing.org), and is a long-time advocate and practitioner of model-based testing.

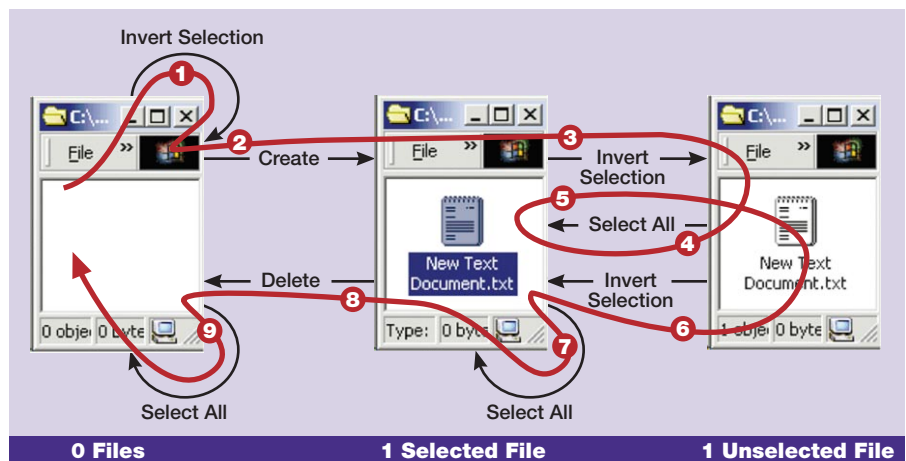


FIGURE 6 A Chinese Postman walk on the new state model

ANNIE BISSETT