Model-Based Testing Using an Implicit State Model

Harry Robinson
Last revised 3 July 2001

Introduction

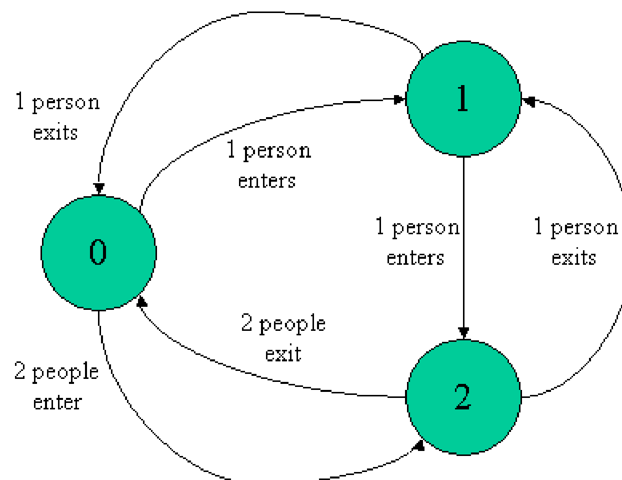
Several people have asked me about whether finite state machines (FSMs) are the only kind of state models that can be used in model-based testing. They usually mention this when we talk about modeling data-centric applications like Notepad, where the finite state model would get really huge really fast if you try to account explicitly for all the strings a user could type.

I think part of the answer to their question lies in whether you choose to model states explicitly or implicitly. This white paper addresses the difference between implicit and explicit state models.

An Explicit State Model

Excellent tools like Test Model Toolkit make it straightforward to create finite state models. And you can use the properties of an explicit state model to drive your testing in very effective, very efficient ways.

Here is a non-technical example. Suppose you are standing outside a room and you want to create a model of how many people are in the room. You cannot see into the room, but you can see when anyone goes in or out. You start off with the assumption that no one is in the room. You could create an explicit state model like the following, where the numbers in the circles indicate how many people are in the room, and the arrows show how we transition from one state to another.



It's easy to see that this model could get cumbersome to maintain by hand as the number of people climbs. One enormous benefit of Test Model Toolkit is that it relieves you of having to maintain this kind of model by hand. Instead you only need to maintain rules for generating the state model.

The advantage of the explicit state model we just created is that we now have an explicit map of the "territory" and can generate test sequences that efficiently cover all possible actions from all possible states. As one example, we could ask this model to generate a minimum length test sequence that uses all possible actions, such as this:

	Action	Ending State
1.	1 person enters	1 person in room
2.	1 person enters	2 people in room
3.	2 people exit	0 people in room
4.	2 people enter	2 people in room
5.	1 person exits	1 person in room
6.	1 person exits	0 people in room

An Implicit State Model

It is also possible to model behavior of a system without explicitly generating all the states as we did in the example above. The resulting implicit model will not be as powerful as the explicit model in generating traversals, but the behavior you are modeling may make it worthwhile to forego some test generation power to make the modeling easier.

Take the room example again. Instead of creating a state model, what if we just keep a counter of how many people are in the room? When people enter the room, we increment the counter by the number of people entering; when people exit the room we decrement our counter by the number exiting. When we need to know how many people are in the room, we simply read the value out of our counter variable. The rules for the model would be the same as in the explicit room model (though we would not take the extra step of generating all the states).

This is a much simpler approach than creating the state graph, though it is not as powerful at generating some types of sequences. For instance, since we do not have an overview of the whole "territory" of our model, we can't easily generate minimum length test sequences that cover all the actions in the model.

There are, however, several very useful ways to generate test sequences on an implicit model:

- Random walks work very well. At each moment in the test, the implicit model evaluates what actions are possible and chooses one of those actions to execute. In the room example above, if the counter indicated that there were no people in the room, then the implicit model would know that it should be impossible for additional people to leave, so the only actions available would be actions that had people entering the room.

- Markov chain-type walks are also good for implicit models. In addition to identifying when actions are possible, these walks allow you to determine how likely it is that the model will choose a particular action. This can help you direct the test sequences into areas that you want to focus on. In the room example, for instance, you might want to focus on what happens when the room is near capacity, so you would make the model more likely to choose actions where people enter the room over actions where they leave the room.

- Finally, the implicit model can go "hybrid" by generating a relatively small number of states in its immediate vicinity and using knowledge of those states to direct its sequences. In communications protocols, this is known as "random state generation". It is more complicated than random walks or Markov chains, but could be significantly more powerful.

So which is the real model: the explicit or the implicit one?

Both!

Both models describe the state space. They differ only in how they use that knowledge about the state space to achieve their goals:

- The explicit model generates the state space so that it can use the relationships among the states and actions to drive its testing.

- The implicit model does not need to generate the states explicitly, so it can model behaviors that might cause an explicit state model to grow explosively.

Tools

As many people in Microsoft know, Test Model Toolkit does a wonderful job handling explicit state models. There is fascinating work underway to understand how TMT can handle the implicit models. For the moment, the example of an implicit model in this paper does not take advantage of TMT.

Implicit Modeling Using a Simple Data Structure

This document and the accompanying Visual Test program describe a simple model of folder creation that can be used for generating tests. There are two purposes to this exercise. First, this exercise shows how we can use modeling even when an explicit state model might be difficult to apply. Second, tree structures such as folders abound in applications that create or delete objects, so this example can serve as a starting point to anyone trying to model a similar function.

The model in this example shows how information about an application can be used to test the application. The model uses a simple data structure in a Visual Test program to maintain information about the state of the application under test.

This model covers the creation of folders and the ability to move through a folder hierarchy. This model does not cover activities such as deleting or renaming folders. Nor does this model allow operations on other kinds of files, such as text files. The extension of this model to cover these activities is straightforward.

The test program

The test program maintains its own data to track the progress of the application it is testing. The program uses Visual Test function calls to interact with the application under test.

For this program, all folders are created underneath a folder called "rootfolder". It doesn't matter where this folder is actually located. The program requires only that the rootfolder window be open on the screen when the program starts and that it should be writable and empty.

The behavior of any particular run of this test program is not pre-determined or scripted. At every point where an action is called for, the Visual Test program consults its internal model to see what actions are available to it. It then randomly chooses one of these actions. The Visual Test program then executes the chosen action.

Because of its internal model, the Visual Test program "knows" what the outcome of its action should be. The bug-finding ability of the model depends on how much information is stored in the internal model and what the test program can determine about the system under test. For instance, the current test program verifies the folder window title and the number of subfolders in the current folder.

The internal model used by the VT program in this discussion is an array that contains the folder names and the index of each folder's parent. For the purposes of this example, this array does not attempt to be efficient or elegant; it only strives to be correct.

This model allows only three actions. The actions are

1. Move up into the parent folder
2. Move down into an existing subfolder
3. Create a new subfolder in the current folder

At any time, any of these actions may be unavailable due to the state of the model. For instance, if there is no existing subfolder, action 2 ("move down into an existing subfolder") is unavailable.

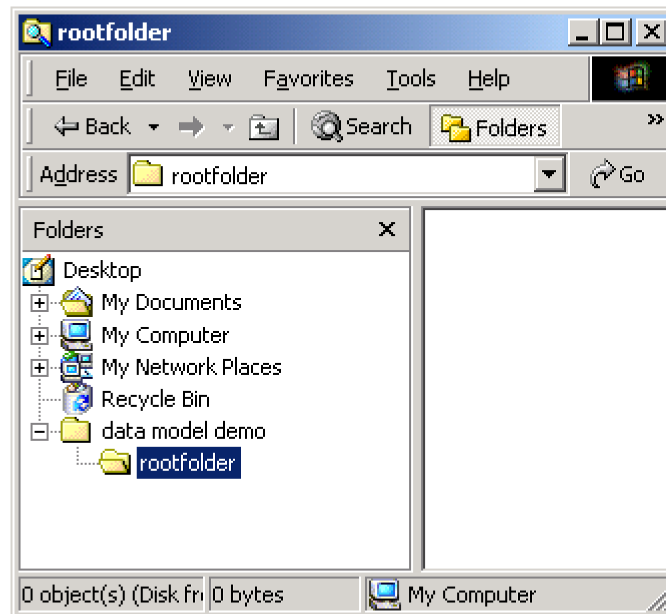
Here is an sample run of the test program.

Step 1 of the Sample Test Run

The model begins in the empty "rootfolder" folder. Rootfolder is the first element in the array (index=0). It has the value –1 for a Parent_Id to indicate that it is the root node for this test.

Index	Parent_Id	File_Name
0	-1	rootfolder

Here is what the folder looks like initially on the screen:



Since rootfolder is the top folder allowed in this test, the program cannot move up to rootfolder's parent folder. Since there are no subfolders yet, the program cannot move down into a subfolder. So, the only action available to the program is to **create a new subfolder.** That folder will be named "f1".

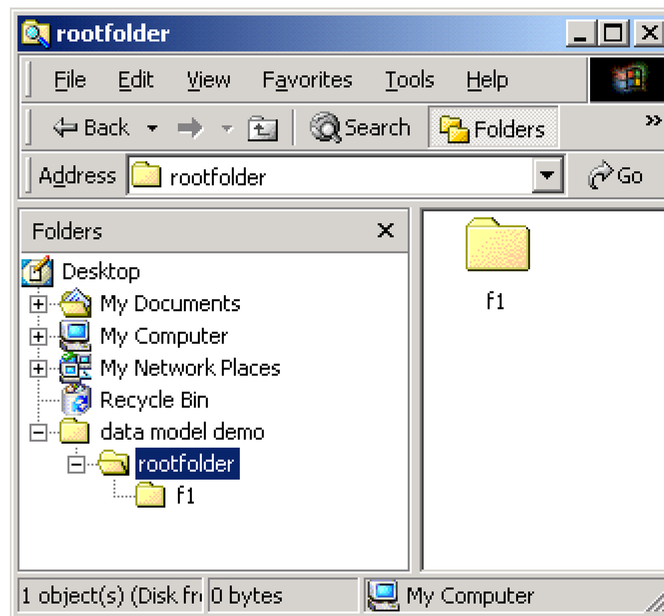
(The naming scheme in this model is very simple. Folders are named "f" followed by their index number from the array. Therefore the first subfolder created in rootfolder is "f1".)

The test program prints out the action it took and the resulting folder layout into the Visual Test viewport. The asterisk after "rootfolder" indicates that rootfolder was the current folder when this action finished. The test program also verifies the current folder name from the window title and it verifies the correct number of subfolders.

```
Action: Create f1 in rootfolder

---------- folder layout ----------
rootfolder*
      f1
-----------------------------------


Current Folder: rootfolder
# immediate subfolders expected:  1
# immediate subfolders found:     1
OK
```

The data array now looks like the table below, showing (by its Parent_Id field) that f1 is a subfolder of rootfolder:

Index	Parent_Id	File_Name
0	-1	rootfolder
1	0	f1



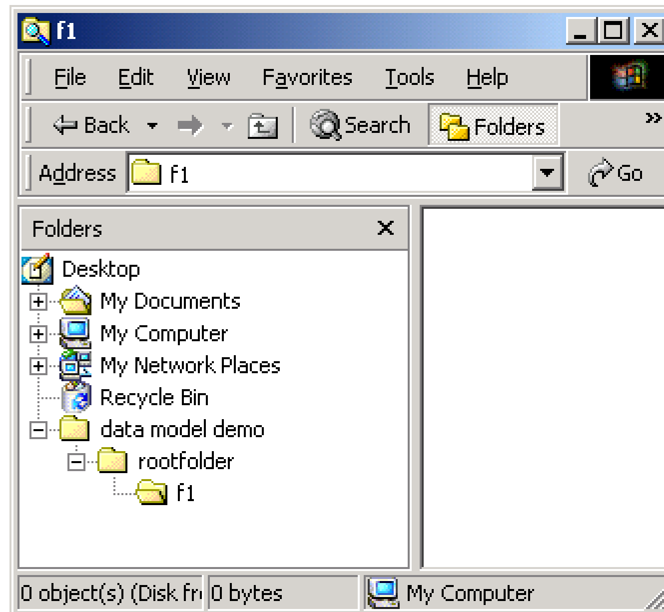
Step 2 of the Sample Test Run

Now that there is a subfolder in rootfolder, the test program can choose to either

- Move down into subfolder f1, or
- Create a new subfolder ("f2") in the current folder

Suppose the test program "decides" to move into subfolder f1:

```
Action: Go down into f1

---------- folder layout ----------
rootfolder
      f1*
-----------------------------------


Current Folder: f1
# immediate subfolders expected:  0
# immediate subfolders found:     0
OK
```



The data array in the test program stays the same, although the variable tracking the position in the folder hierarchy tells us we are now in subfolder f1.

Step 3 of the Sample Test Run

The test program is now in subfolder f1. The parent folder or f1 is rootfolder. The test program can either

- Move up into the parent folder, or
- Create a new subfolder ("f2") in the current folder

(Note that the folder "f2" was not created in the previous action.)

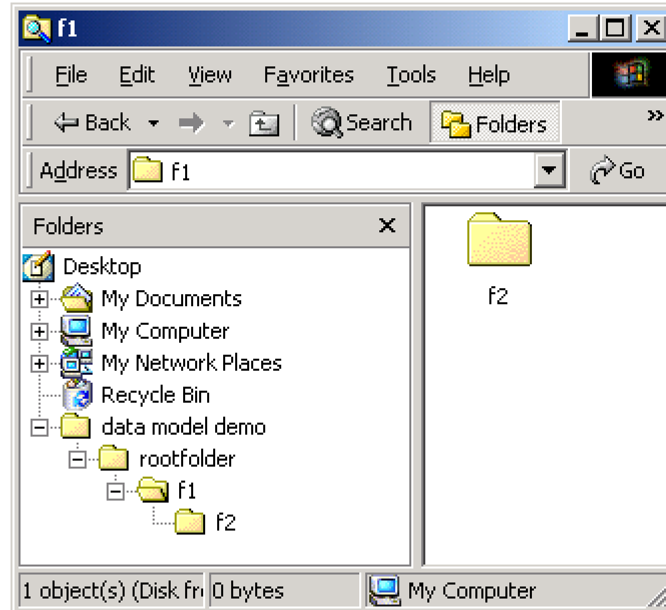
Suppose the test program "decides" to create the subfolder "f2" in f1

```
Action: Create f2 in f1

---------- folder layout ----------
rootfolder
      f1*
            f2
-----------------------------------


Current Folder: f1
# immediate subfolders expected:  1
# immediate subfolders found:     1
OK
```

The internal model now looks like this, showing that f2 is a subfolder of f1:

Index	Parent_Id	File_Name
0	-1	rootfolder
1	0	f1
2	1	f2

And here is what it looks like on the screen:



The test program can be set to create an arbitrary number of folders (within some MAX_PATH limit). The actual distribution of folders in the test run is random because the test program can move around in the hierarchy and chooses its actions randomly. Therefore, every run of the test program is likely to generate a different test.

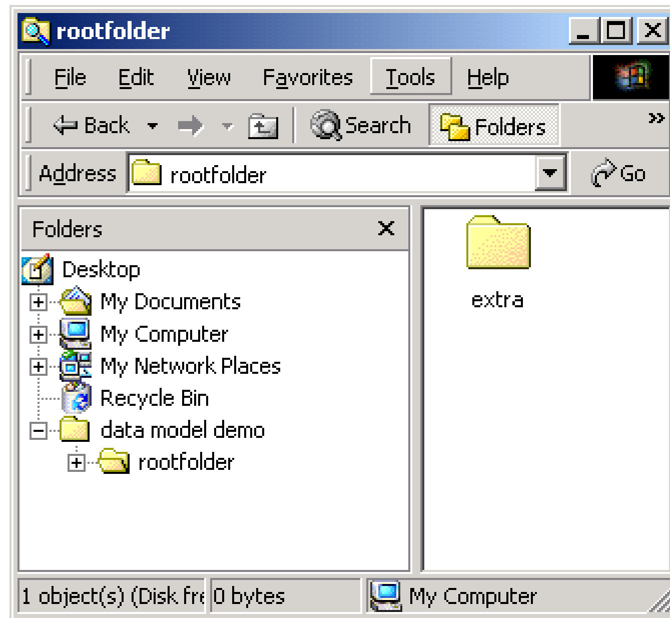
[**Disclaime**r: this Visual Test program does not clean up after itself, so you will need to remove all subfolders of rootfolder before re-running the program.]

Detecting a bug

How can this test program detect a bug in the behavior of the folder creation? Let's stage a pseudo-bug to demonstrate.

Our test assumes that rootfolder is empty at the start of the test run. If we plant an extra folder in the rootfolder before starting the test, the test program should detect that its count of subfolders is incorrect.

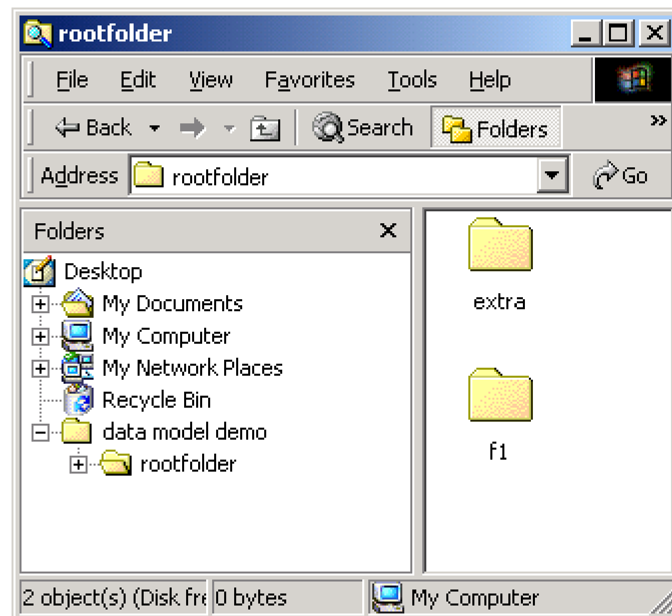
Here is rootfolder before the test starts. I have put a subfolder named "extra" into rootfolder:



After the test program executes its first action, the test oracle routine detects that the subfolder count is wrong:

```
Action: Create f1 in rootfolder

---------- folder layout ----------
rootfolder*
      f1
------------------------------------


Current Folder: rootfolder
# immediate subfolders expected:  1
# immediate subfolders found:     2
Error!
```

And here is what rootfolder looks like on the screen:



So what does this implicit modeling buy you?

- As with explicit modeling, modeling the manipulation of folders this way means that you can leave the computer to run tests for you unattended. The tests it runs will not be merely regression tests; they will constantly choose new paths.

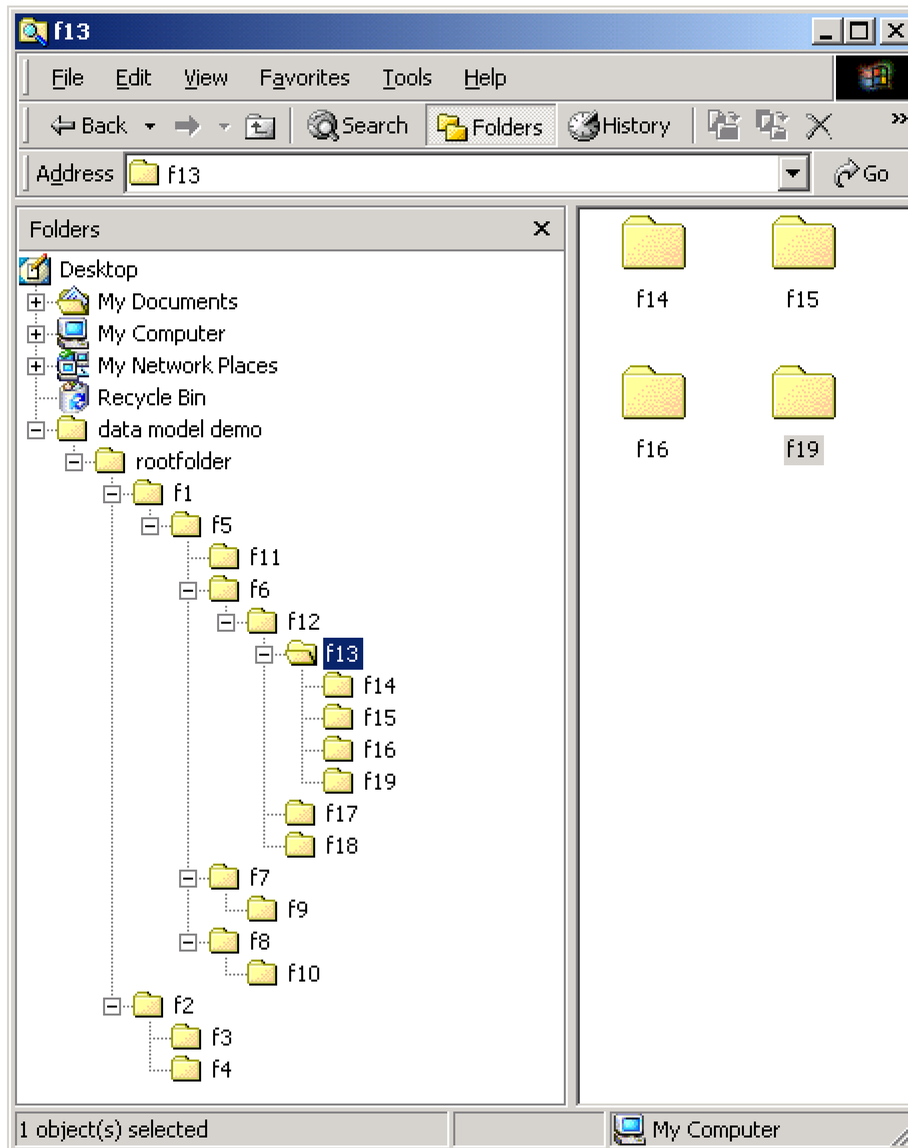
- You can set arbitrary limits on the number and type of folders and files and how those folders and files are manipulated.

- The model allows new functionality (such as deleting of folders) to leverage the existing testing.

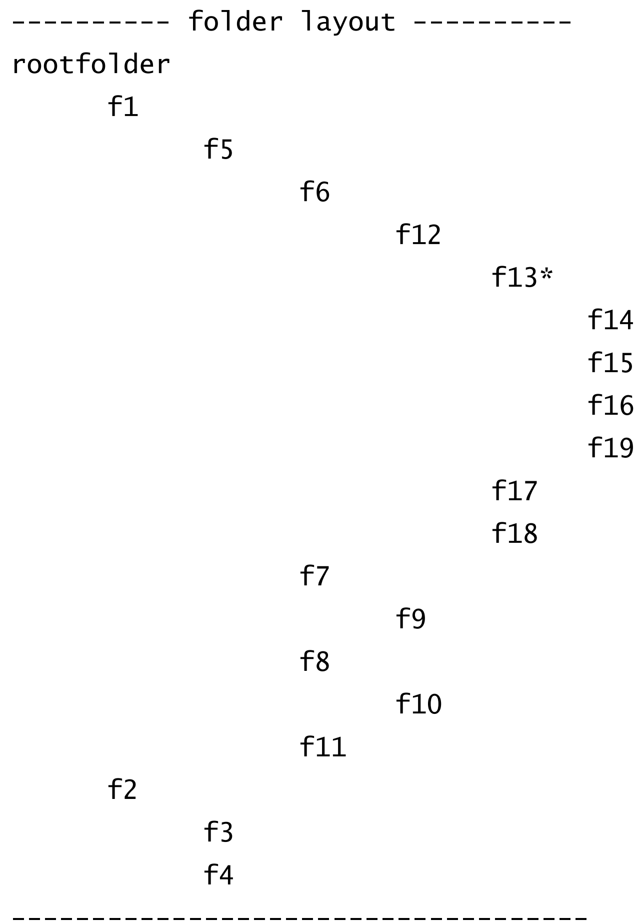
- Because the test program "knows" what is supposed to happen in the application, it is more powerful than a "dumb monkey" program. A model-based test program can find non-crashing bugs (such as incorrect subfolder counts) as well as crashing bugs.

Appendix:

As eye candy, here is an example of what happened when the test program using an implicit state model created 19 subfolders on a recent run:



And here is what the internal model expected the folder layout to look like after creating those 19 subfolders in various places in the hierarchy. (The asterisk indicates that subfolder f13 was the current directory.)



```
---------- folder layout ----------
rootfolder
      f1
            f5
                  f6
                        f12
                              f13*
                                    f14
                                    f15
                                    f16
                                    f19
                              f17
                              f18
                  f7
                        f9
                  f8
                        f10
                  f11
      f2
            f3
            f4
-----------------------------------
```

If you look carefully, you will see that the model and the actual state of the application are in sync.

Acknowledgment
My thanks to Noel Nyman for his excellent feedback on an earlier draft of this paper.