Presentation Notes
Paper
Bio
Return to Main Menu

PRESENTATION

W5

Wednesday, November 3, 1999
1:00 PM

FINITE STATE MODEL-BASED TESTING ON A SHOESTRING

Harry Robinson
Microsoft Corporation

INTERNATIONAL CONFERENCE ON
SOFTWARE TESTING, ANALYSIS & REVIEW
NOVEMBER 1-5, 1999
SAN JOSE, CA

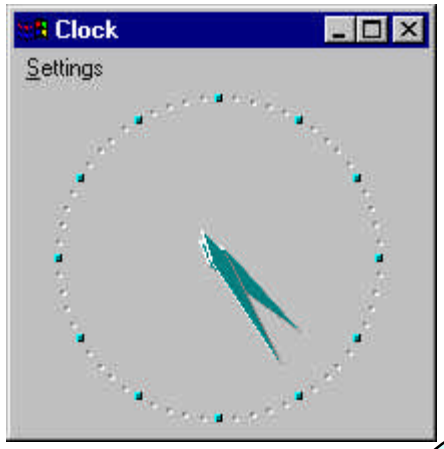
Finite State Model-Based Testing on a Shoestring

Harry Robinson
Intelligent Search Test Group
Microsoft

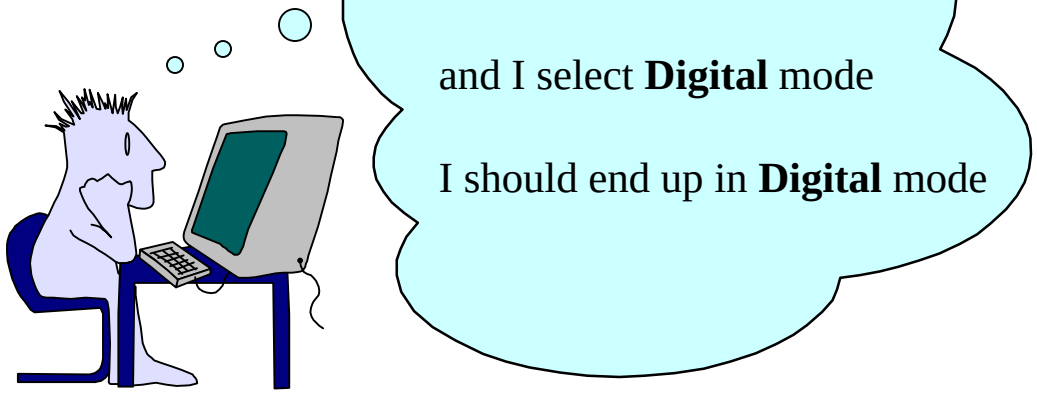
So What's a Model?

- A model is a description of a system's behavior.
- Models are simpler than the systems they describe.
- Models help us understand and predict the system's behavior.

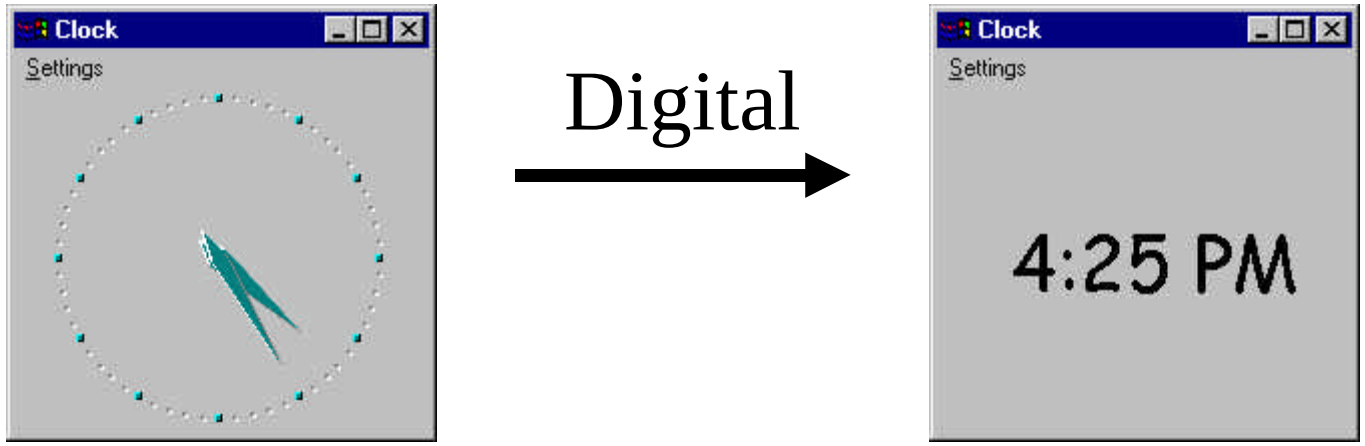
We All Use Models Already



Digital



How to Use Models in Testing



Setup: Clock is in Analog mode

Action: Select Digital mode

Outcome: Did Clock go correctly to Digital mode?

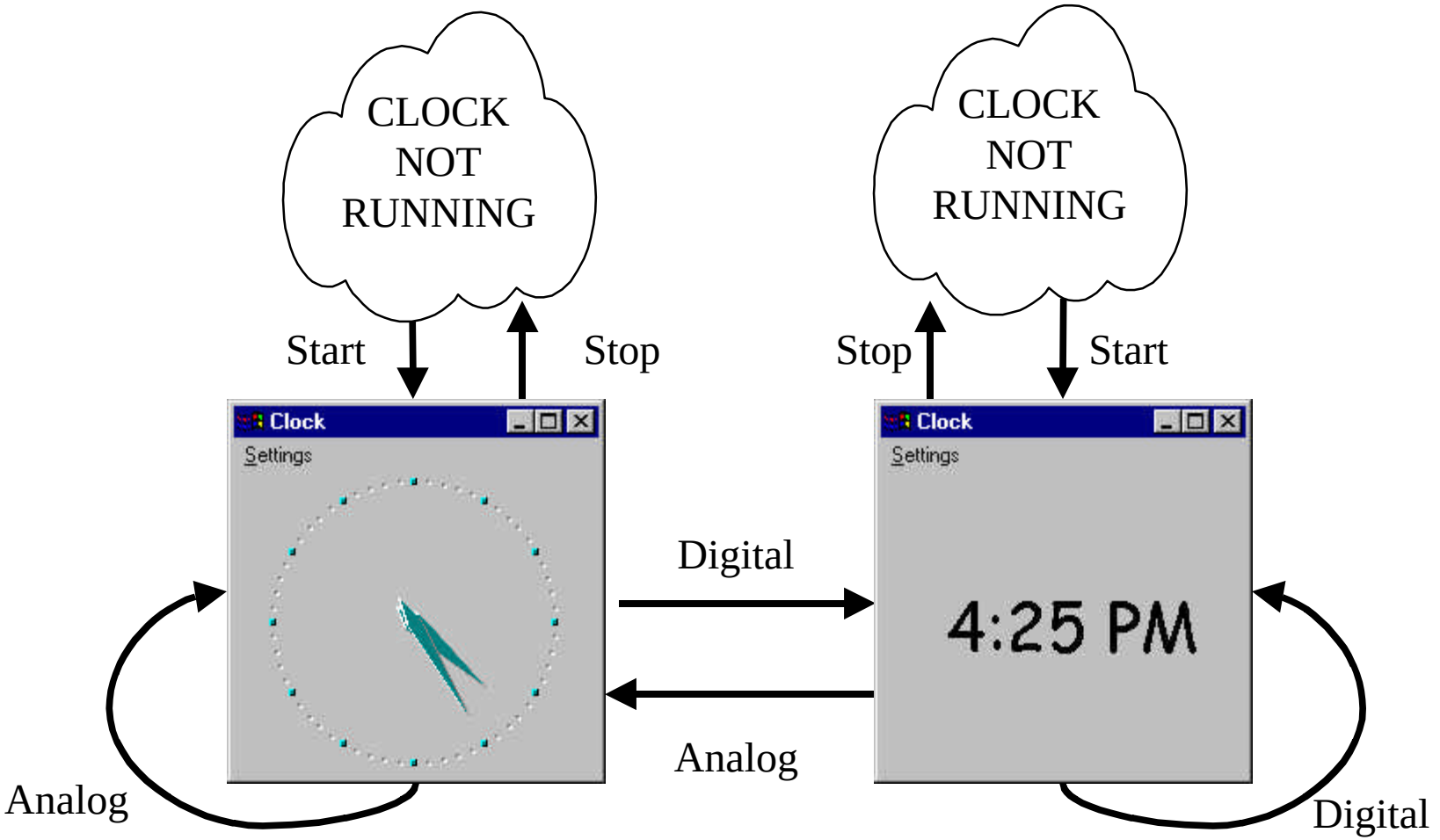
How to Create Model-Based Tests

- Create a state model of the application
- Generate sequences of test actions
- Execute the test actions
- Determine if the application worked right
- Find bugs

Step 1:

Create a state model of the application

A Simple Clock State Model



All Actions Aren't Always Available

System Mode = NOT RUNNING

Action = Stop

Rule: You can't execute the Stop action if the Clock is not running

Operational Modes

Operational modes are state attributes that determine

- the actions that are possible in a state, and
- what outcome will result if an action is taken.

Example:

- Clock is running; Available actions: Analog, Digital, Stop
- Clock is not running; Available actions: Start

Using Rules to Build the Model

Stop

- If the Clock is not running, the user cannot execute the **Stop** action.
- If the Clock is running, the user can execute the **Stop** action.
- After the **Stop** action executes, the application is not running

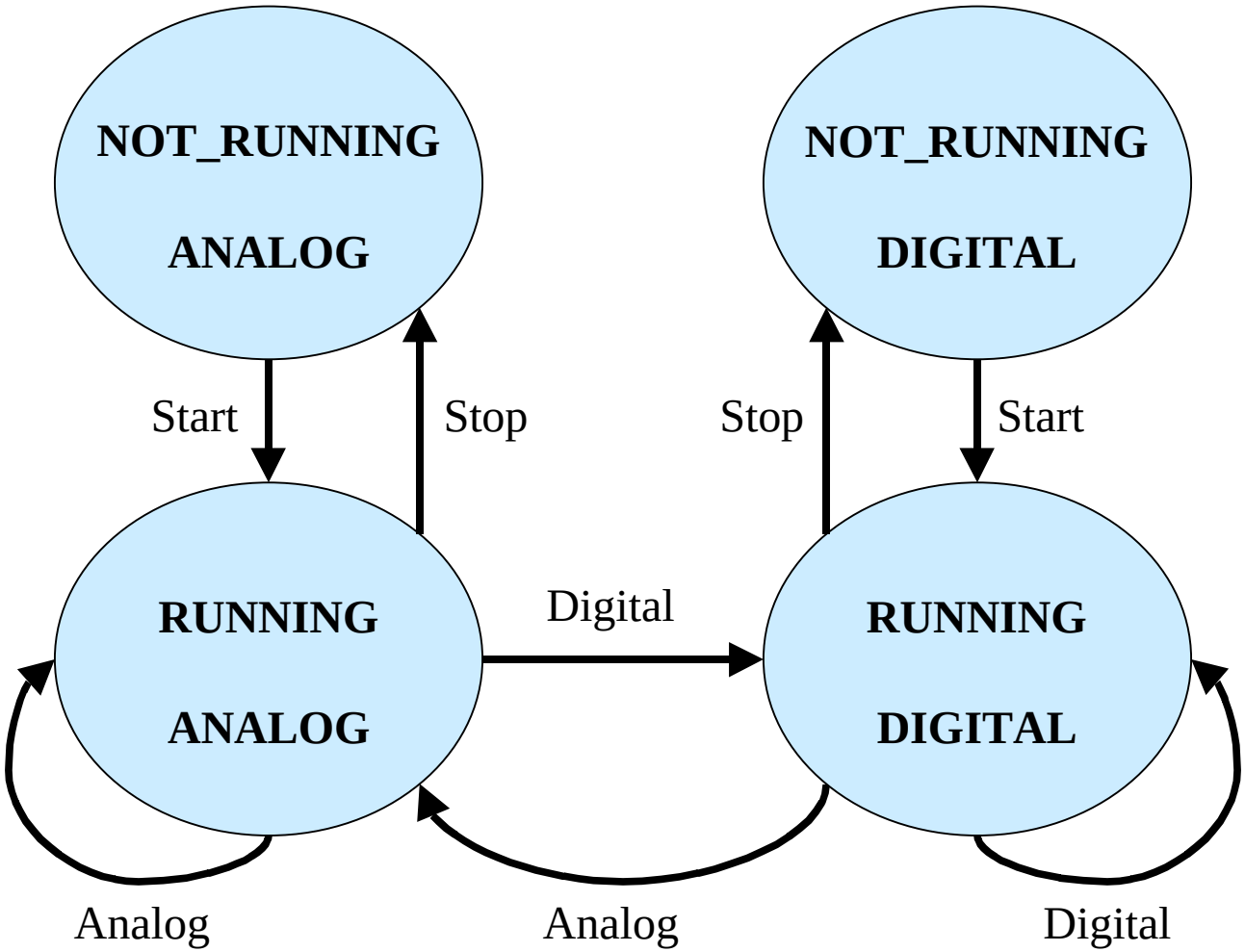
Using VT Code to Build the Model

```
possible = TRUE                                  ‘ assume the action is possible
if (action = “Stop” ) then                       ‘ want to do a Stop action?
    if (system_mode = RUNNING) then              ‘ if clock is in running mode
        new_system_mode = NOT_RUNNING            ‘ clock goes to not running mode
    else                                         ‘ otherwise
        possible = FALSE                         ‘ Stop action is not possible
    endif
endif

if (possible = TRUE) then                        ‘ if action is possible
    print system_mode;”.”;setting_mode,          ‘ print beginning state
    print action,                                ‘ print the test action
    print new_system_mode;”.”;new_setting_mode   ‘ print ending state
endif
```

The Generated Finite State Table

Beginning State	Action	Ending State
NOT_RUNNING.ANALOG	Start	RUNNING.ANALOG
NOT_RUNNING.DIGITAL	Start	RUNNING.DIGITAL
RUNNING.ANALOG	Stop	NOT_RUNNING.ANALOG
RUNNING.DIGITAL	Stop	NOT_RUNNING.DIGITAL
RUNNING.ANALOG	Analog	RUNNING.ANALOG
RUNNING.ANALOG	Digital	RUNNING.DIGITAL
RUNNING.DIGITAL	Analog	RUNNING.ANALOG
RUNNING.DIGITAL	Digital	RUNNING.DIGITAL

The Clock State Model (rephrased)



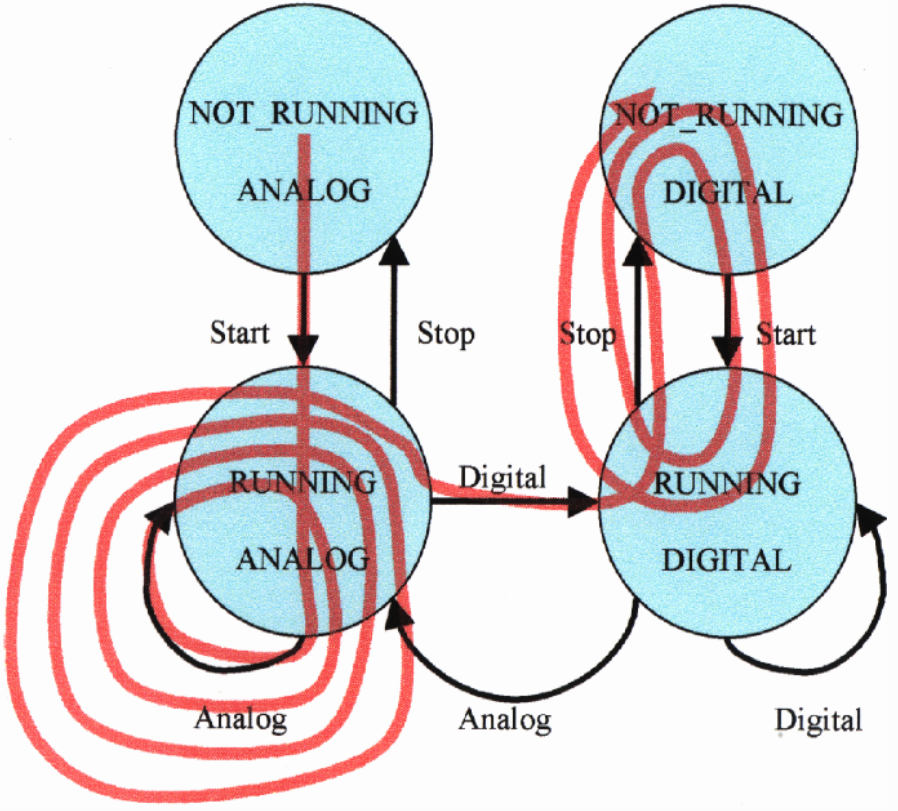
Step 2:

Generate sequences of test actions

Generating Test Sequence 1

Random Walk

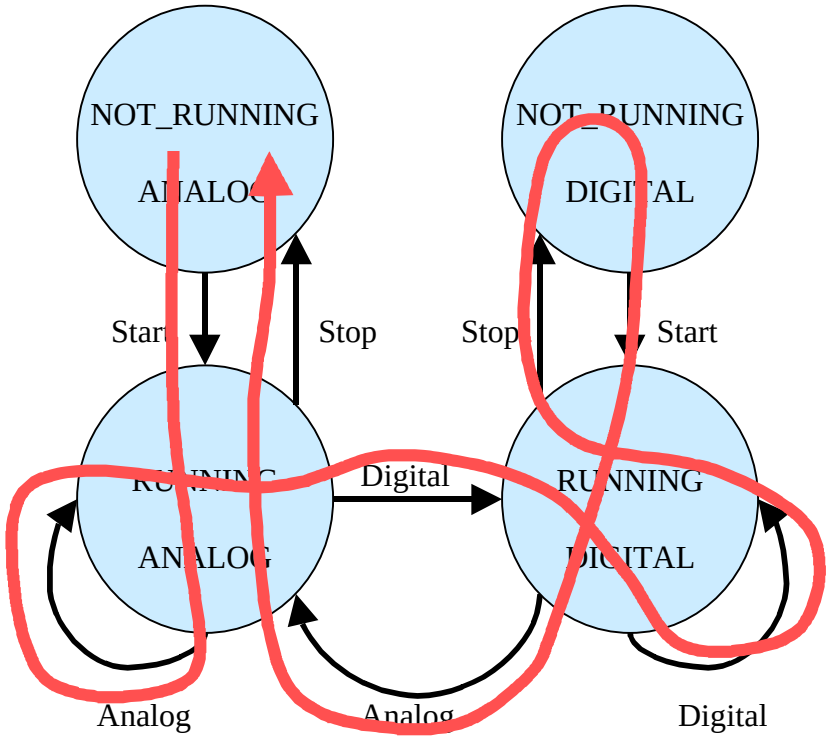
Start
Analog
Analog
Analog
Analog
Analog
Digital
Stop
Start
Stop
Start
Stop



Generating Test Sequence 2

Chinese Postman

Start
Analog
Digital
Digital
Stop
Start
Analog
Stop



Step 3:

Execute the test actions

Visual Test functions

`Run("C:\WINNT\System32\clock.exe")`	Starts the Clock application
`WMenuSelect( "Settings\Analog")`	Chooses the menu item "Analog" on the "Settings" menu
`WSysMenu( 0 )`	Brings up the System menu for the active window
`WFndWnd("Clock")`	Finds an application window with the caption "Clock"
`WMenuChecked("Settings\Analog")`	Returns TRUE if menu item "Analog" is check-marked
`GetText(0)`	Returns the window title of the active window

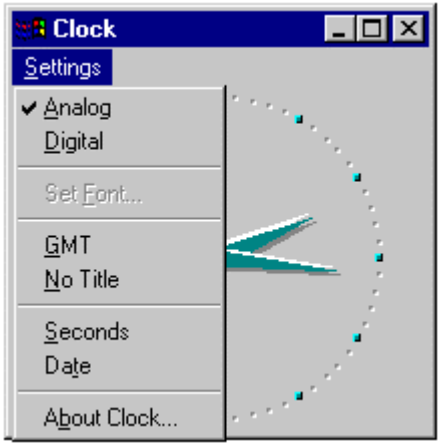
Executing the Test Actions

```
open "test_sequence.txt" for input as #infile     ‘get the list of test actions

while not (EOF(infile))

    line input #infile, action                    ‘read in a test action

    select case action

    case “Start“                                  ‘ Start the Clock
        run("C:\WINNT\System32\clock.exe”)        ‘ VT call to start clock

    case “Analog“                                 ‘ choose Analog mode
        WMenuSelect("Settings\Analog")            ‘ VT call to select Analog

    case “Digital“                                ‘ choose Digital mode
        WMenuSelect("Settings\Digital")           ‘ VT call to select Digital

    case “Stop“                                   ‘ Stop the Clock
        WSysMenu (0)                              ‘ VT call to bring up system menu
        WMenuSelect ("Close")                     ‘ VT call to select Close

    end select

wend
```

Step 4:

Determine if the application worked right

Use Rules as Test Oracles



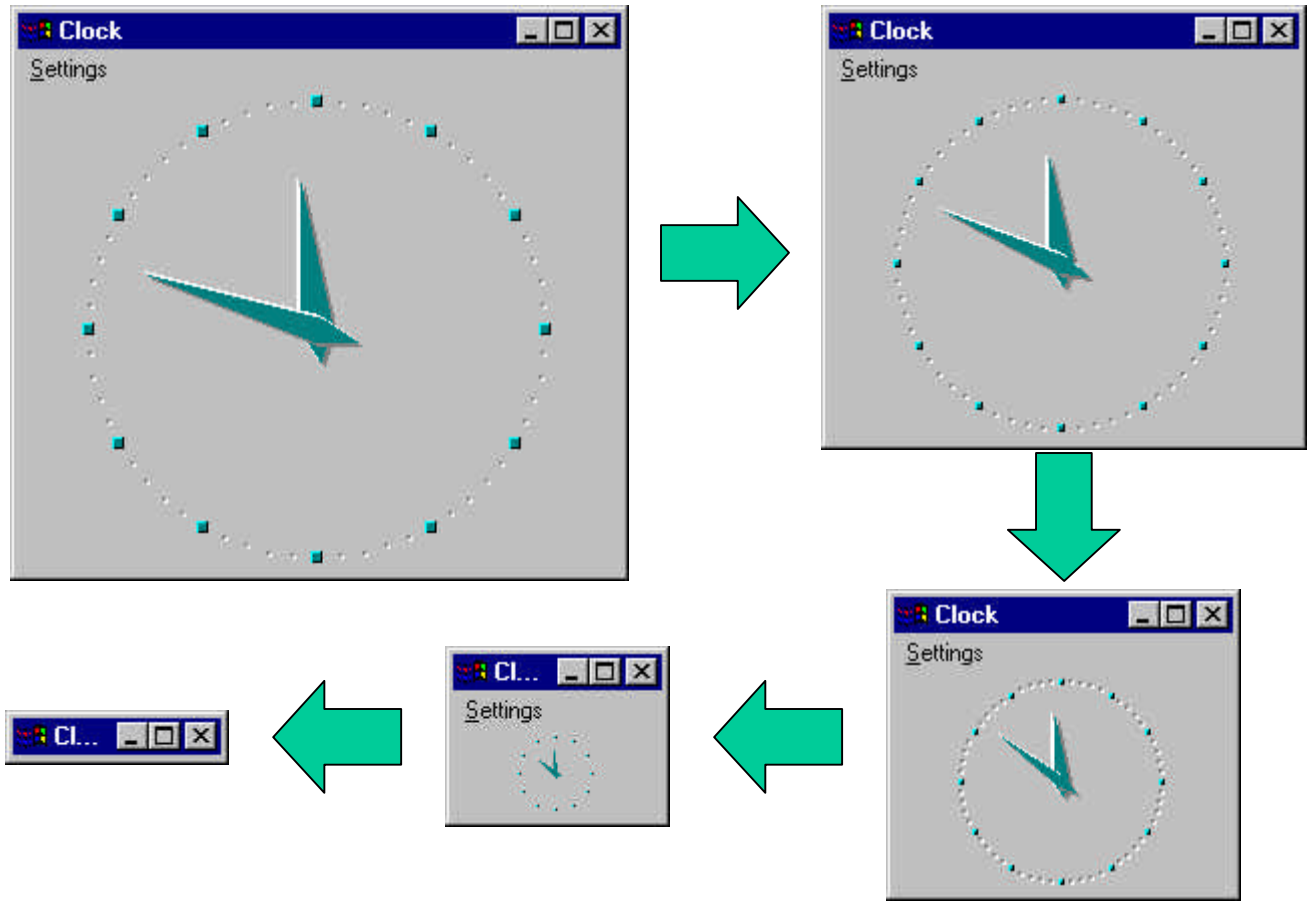

```
if ( (setting_mode = ANALOG) _                    'if we are in Analog mode
AND NOT WMenuChecked("Settings\Analog") ) then    'but Analog is not check-marked
        print "Error: Clock should be Analog mode"  'alert the tester
        stop
endif
```

Step 5:

Find bugs

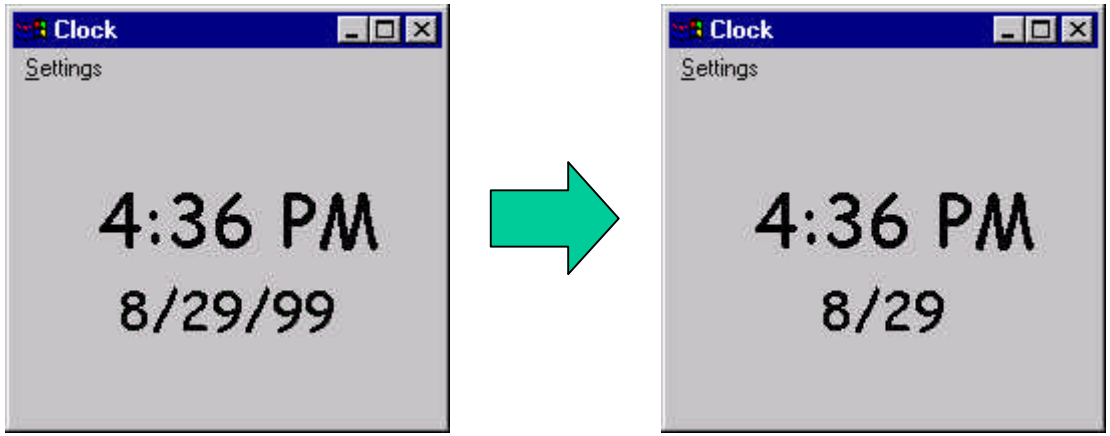
The Incredible Shrinking Clock

Start
Maximize
Stop
Start
Minimize
Stop
Start
Restore
Stop



Where Have the Years Gone?

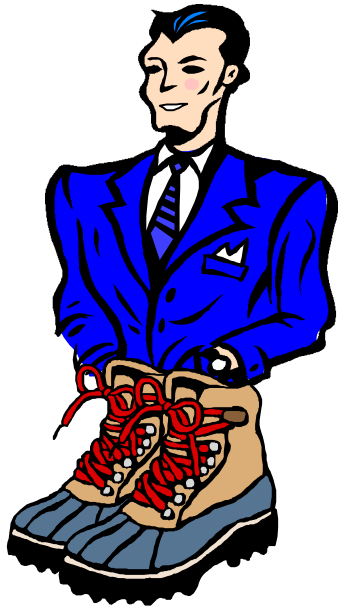
Start
Minimize
Stop
Start
Restore
Date



Conclusions

Model-based testing is

- powerful
- flexible
- maintainable
- low-cost!



For more info …

www.model-based-test.org

Thank you!

Finite State Model-Based Testing on a Shoestring

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

Abstract

Model-based testing is a software test technique that generates tests from an explicit model of software behavior. Modern programmable test tools allow us to use this technique to create useful, flexible and powerful tests at a very reasonable cost.

What Is Model-Based Testing?

Model-based testing is a technique that generates software tests from explicit descriptions of an application's behavior. Creating and maintaining a model of an application makes it easier to generate and update tests for that application.

Several good model-based test tools are currently available in the market, but the techniques of model-based testing are not tied to any tool. This paper shows how anyone willing to do some test programming can implement model-based testing in low-cost test language tools. The test language used in this paper is Visual Test [1] from Rational Software.

In this paper, I will discuss how to use a test programming language to

1. Create a finite state model of an application.
2. Generate sequences of test actions from the model.
3. Execute the test actions against the application.
4. Determine if the application worked right.
5. Find bugs.

What Is A Model?

A model is a description of a system's behavior. Because models are simpler than the systems they describe, they can help us understand and predict the system's behavior.

State models are common in computing and have been shown to be a useful way to think about software behavior and testing [2][3]. A finite state model consists of a set of states, a set of input events and the relations between them. Given a current state and an input event you can determine the next current state of the model.

As a running example throughout this paper, we will create a simple finite state model of the Windows NT Clock application [4]. The Clock can be found as Programs\Accessories\Clock under the Start menu in Windows NT.

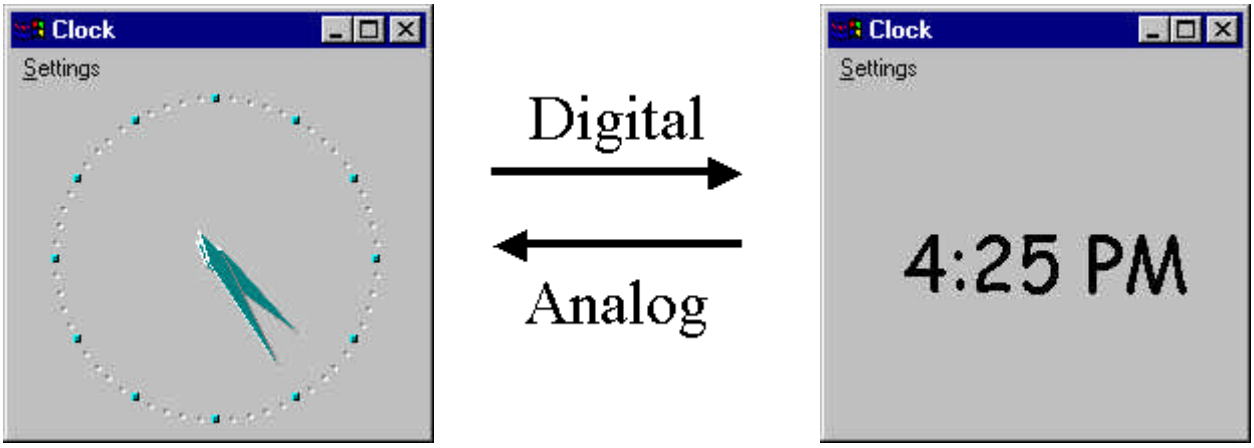


Figure 1: A Very Simple Finite State Model of the Clock

Figure 1 shows two forms of the Clock display. The left side shows the Analog display; the right side shows the Digital display. If the Clock application is in the Analog display mode, clicking the menu selection "Settings\Digital" moves the application into the Digital display. Likewise, if the application is in the Digital display mode, clicking the menu selection "Settings\Analog" moves the application into the Analog display.

We could use this very simple state model as a basis for tests, where following a path in the model is equivalent to running a test:

Setup:	Put the Clock into its Analog display mode
Action:	Click on "Settings\Digital"
Outcome:	Does the Clock correctly change to the Digital display?

Create a Finite State Model of an Application

Finite state models are excellent tools for understanding and testing software applications. However, a very large state model is needed to describe a complex system in enough detail to do a good job testing. A finite state model used in representing the behavior of an application is likely to have many, many states – so many that it would be tedious and unrealistic to create and maintain the model by hand.

The approach advocated in this paper allows you to generate large state models by describing the behavior of an application in terms of a small number of state attributes called operational modes [5]. Operational modes are the attributes of a state that determine what user actions are possible in that state and what outcomes will occur when actions are executed. For instance, whether or not the application is currently running is a common operational mode. Typically, if the application is NOT running, the only action the user can execute is to start the application. On the other hand, if the application IS running, the user has a much greater choice of actions that could be performed.

For the purposes of this paper, we will only be concerned with the following actions in the Clock:

- **Start** the Clock application
- **Stop** the Clock application
- Select **Analog** setting
- Select **Digital** setting.

The rules for these actions in the Clock application are as follows:

- **Start**
  - If the application is NOT running, the user can execute the **Start** command.
  - If the application is running, the user cannot execute the **Start** command.
  - After the **Start** command executes, the application is running.
- **Stop**
  - If the application is NOT running, the user cannot execute the **Stop** command.
  - If the application is running, the user can execute the **Stop** command.
  - After the **Stop** command executes, the application is not running.
- **Analog**
  - If the application is NOT running, the user cannot execute the **Analog** command.
  - If the application is running, the user can execute the **Analog** command.
  - After the **Analog** command executes, the application is in Analog display mode.
- **Digital**
  - If the application is NOT running, the user cannot execute the **Digital** command.
  - If the application is running, the user can execute the **Digital** command.
  - After the **Digital** command executes, the application is in Digital display mode.

Our model in this example will have two operational modes, system mode and setting mode, which can have the following values:

- System mode: NOT_RUNNING means Clock is not running
RUNNING means Clock is running
- Setting mode: ANALOG means Analog display is set
DIGITAL means Digital display is set

We now have the actions, rules and operational modes for our model. We can use our test programming language to create a list of the state transitions in the finite state model. We will simply run through all possible combinations of operational mode values and print out any possible transitions.

The names of the states will appear as a list of operational mode values separated by periods. For instance, the state "RUNNING.DIGITAL" means the application is running and the display is in Digital mode.

Here is the Visual Test code you would write to generate the model's states and transitions:

```
for system_mode = NOT_RUNNING to RUNNING                    ‘ for all system modes

        for setting_mode = ANALOG to DIGITAL                ‘ for all setting modes

                for action = Start to Digital               ‘ actions: Start,Stop,Analog,Digital

                        possible = TRUE                     ‘ assume action is possible
                        new_system_mode = system_mode       ‘ assume mode values do not change
                        new_setting_mode = setting_mode

                        select case action

                        case “Start”                                    ‘start the clock
                                if (system_mode = NOT_RUNNING) then     ‘clock must be NOT running
                                        new_system_mode = RUNNING       ‘clock goes to running
                                else
                                        possible = FALSE
                                endif

                        case “Stop”                                     ‘stop the clock
                                if (system_mode = RUNNING) then         ‘clock must be running
                                        new_system_mode = NOT_RUNNING ‘clock goes to NOT running
                                else
                                        possible = FALSE
                                endif


                        case “Analog”                                   ‘choose analog mode
                                if (system_mode = RUNNING) then         ‘clock must be running
                                        new_setting_mode = ANALOG       ‘clock goes to analog mode
                                else
                                        possible = FALSE
                                endif

                        case “Digital”                                  ‘choose digital mode
                                if (system_mode = RUNNING) then         ‘clock must be running
                                        new_setting_mode = DIGITAL      ‘clock goes to digital mode
                                else
                                        possible = FALSE
                                endif

                        end select

                        if (possible = TRUE) then                               ‘ if action possible
                                print system_mode;”.”;setting_mode,             ‘ print begin state
                                print action,                                   ‘ print action
                                print new_system_mode;”.”;new_setting_mode      ‘ print end state
                        endif

                next action

        next setting_mode

next system_mode
```

The state table printed by the code above after running through the values of the operational modes looks as follows:

Beginning State	**Action**	**Ending State**
NOT_RUNNING.ANALOG	Start	RUNNING.ANALOG
NOT_RUNNING.DIGITAL	Start	RUNNING.DIGITAL
RUNNING.ANALOG	Stop	NOT_RUNNING.ANALOG
RUNNING.DIGITAL	Stop	NOT_RUNNING.DIGITAL
RUNNING.ANALOG	Analog	RUNNING.ANALOG
RUNNING.ANALOG	Digital	RUNNING.DIGITAL
RUNNING.DIGITAL	Analog	RUNNING.ANALOG
RUNNING.DIGITAL	Digital	RUNNING.DIGITAL

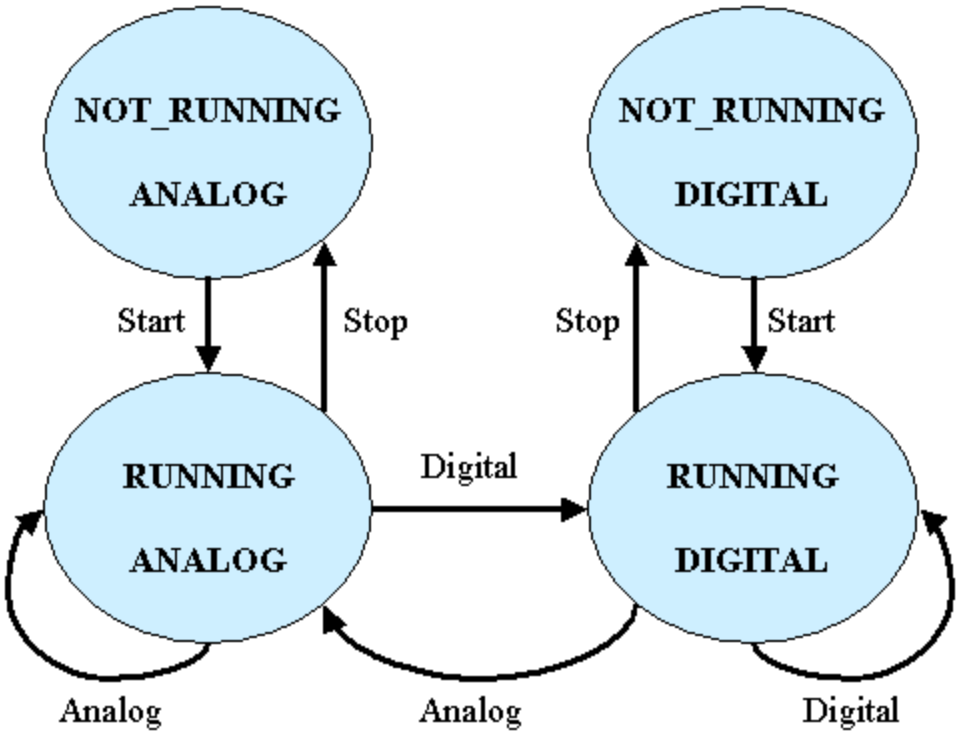


Figure 2: State Transition Diagram for the Clock Model

Figure 2 shows a graphic representation of the finite state model. The circles represent states and the arcs represent actions. We can see how running the Start command from the "NOT_RUNNING.ANALOG" state brings the application into the "RUNNING.ANALOG" state. We also note with interest that it is possible to run the Analog command while in the "RUNNING.ANALOG" state; the arc loops back to "RUNNING.ANALOG", meaning that the application ends up in the same state it started from.

Generate Sequences of Test Actions from the Model

As we pointed out with Figure 1, testing an application is like following a path through a finite state model. Now that we have our state model constructed, we can use various techniques to choose what paths we want our tests to take through it.

One of the most popular choices is to allow the tests to move randomly through the state model, taking any available action out of a state. Given enough time, these random walks can cover a good part of the application. The random nature of their choices means that they tend to produce unusual combinations of actions that human testers wouldn't bother to try, such as

```
Start
Analog
Analog
Analog
Analog
Analog
Stop
```

A more advanced path generation technique, called a “Chinese Postman tour”, touches every action in the state model as efficiently as possible. (For more information on generating paths through a finite state model, see [6].) A Chinese Postman tour on our simple state model might look as follows:

```
Start
Analog
Digital
Digital
Stop
Start
Analog
Stop
```

These sequences of actions can be stored in an external file (such as “test_sequence.txt” in this case). This action sequence file then serves as the instructions to the test execution phase.

Execute the Test Actions Against the Application

Visual Test has a rich set of functions for interacting with the application you are testing. The table below lists some examples of these functions and a description of what they do.

`Run(“C:\WINNT\System32\clock.exe”)`	Starts the Clock application
`WMenuSelect( “Settings\Analog”)`	Chooses the menu item “Analog” on the “Settings” menu
`WSysMenu( 0 )`	Brings up the System menu for the active window
`WFndWnd("Clock")`	Finds an application window with the caption “Clock”
`WMenuChecked("Settings\Analog")`	Returns TRUE if menu item “Analog” is checkmarked
`GetText(0)`	Returns the window title of the active window

The test execution phase is kept deliberately simple. The program listed below reads in a list of actions from a file, executes the function associated with that action, and then reads the next action from the file.

As an example, suppose the first two actions in “test_sequence.txt” are “Start” and “Analog”. Our test execution program would read “Start” from the file and execute the Visual Test function `run("C:\WINNT\System32\clock.exe”)` associated with “Start” action. Our program would then read “Analog” from the file and execute the `WMenuSelect("Settings\Analog")` function associated with the “Analog” action.

```
open "test_sequence.txt" for input as #infile       ‘get the list of actions

while not (EOF(infile))

        line input #infile, action                  ‘read in an action

        select case action

        case "Start"                                ‘start the Clock
                run("C:\WINNT\System32\clock.exe”)  ‘VT call to start Clock program
```

```
        case "Analog"                                   'choose analog mode
               WMenuSelect("Settings\Analog")           'VT call to select menu item Analog

        case "Digital"                                  'choose digital mode
               WMenuSelect("Settings\Digital")          'VT call to select menu item Digital

        case "Stop"                                     'stop the Clock
               WSysMenu (0)                             'VT call to bring up system menu
               WMenuSelect ("Close")                    'VT call to select menu item Close

        End select

        Test_oracle()                           'determine if Clock behaved correctly

wend
```

After each action is executed, a test oracle function, described in the next section, is called to determine if the application behaved as the model expected.

Determine if the Application Worked Right

A test oracle is a mechanism that verifies if the application has behaved correctly. One of the great benefits of model-based testing is the ability to create a test oracle from the state model.

In the case of our simple Clock model, we would like to verify whether the Clock is running or not, and we would like to verify whether we are in Analog or Digital display mode.

To determine if Clock is running, we can use the WFndWnd("Clock") function call in Visual Test to look for the application window captioned "Clock".

Finding out whether the Analog or Digital display is showing is slightly trickier. Since both the Analog and Digital clock faces are images that cannot easily be interpreted by the test program, we will resort to some secondary characteristics of the application.

If you look carefully at the two "Settings" menus displayed in Figure 3, you will see that the menu item for the current display mode has a checkmark next to it. We can use this information for our oracle. If we are in Analog display mode and we bring up the "Settings" menu, we will expect to see a checkmark next to the word "Analog"; otherwise, we must be in Digital display mode and we will expect to see a checkmark next to the word "Digital". This method is not as direct as looking at the display (as a person could) but it allows us to compensate for what the test program cannot see.

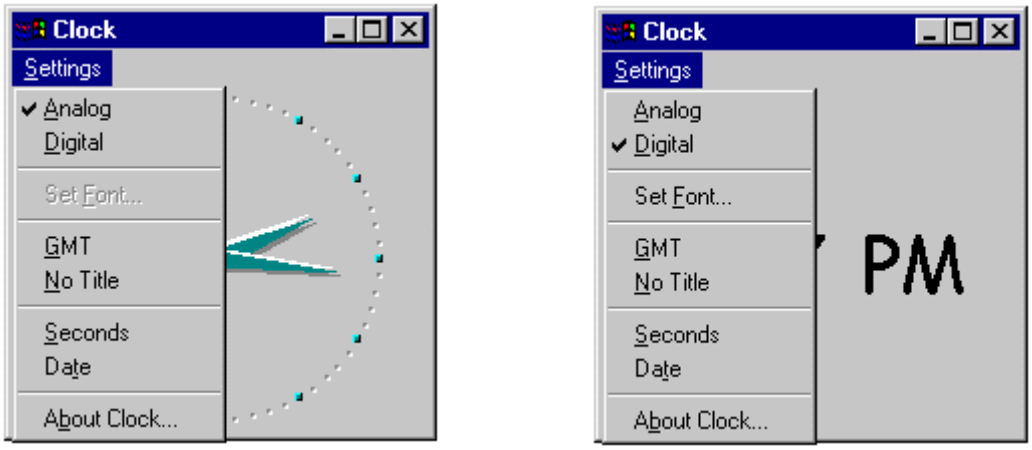


Figure 3: The menu checkmarks indicate whether the Clock is in Analog or Digital display mode

Here is Visual Test code that implements the test oracle for the Analog/Digital display mode:

```
if (system_mode = RUNNING) then

        if ( WFndWnd("Clock") = 0 ) then                    'if no "Clock" running
                print "Error: Clock should be Running"      'print the error
                stop
        endif

        if  ( (setting_mode = ANALOG) _                     'if analog mode
        AND NOT WMenuChecked("Settings\Analog") ) then      'but no check next to Analog

                print "Error: Clock should be Analog mode"  'print the error
                stop

        elseif  ( (setting_mode = DIGITAL) _                'if digital mode
        AND NOT WMenuChecked("Settings\Digital") ) then     'but no check next to Digital

                print "Error: Clock should be Digital mode" 'print the error
                stop

        endif

endif
```

Find Bugs

Testing is about finding bugs. Model-based testing finds it bugs by executing the application and verifying the results against the state model. When discrepancies between the application and the model are detected, the test program alerts the tester.

Here are two interesting bugs that surfaced in the model-based testing of Clock. The first bug (Figure 4) was detected when the tests were no longer able to find the clock face. Some investigative work turned up that the problem occurred when the sequence to the left of the figure was executed. Every time this sequence was executed, the Clock window would reappear a few pixels smaller.

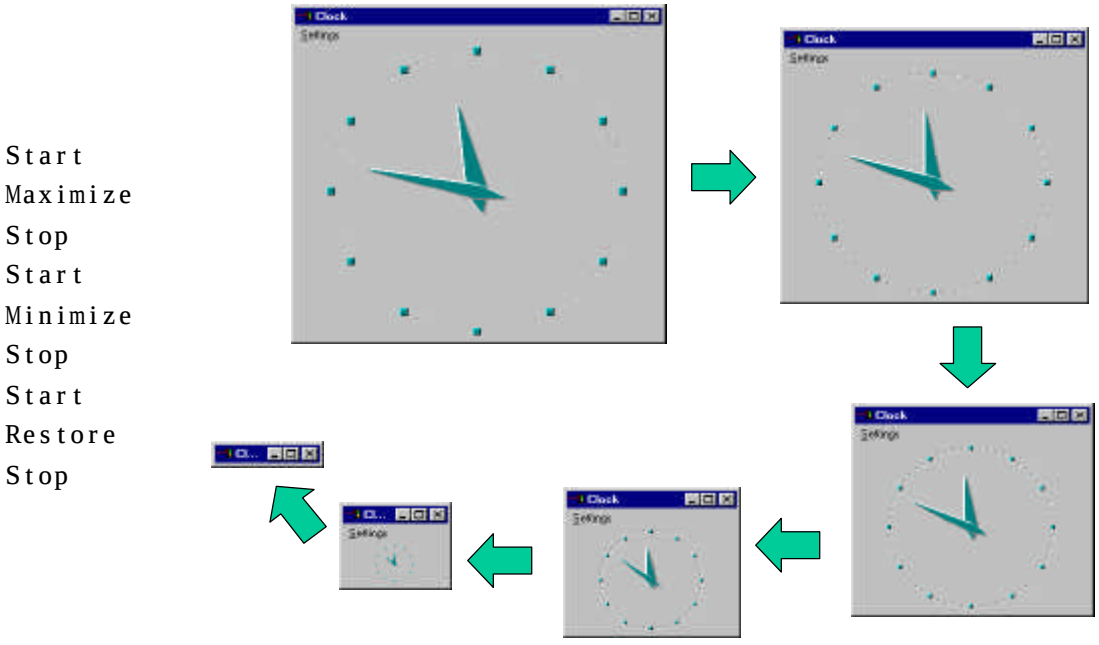


Figure 4: The Incredible Shrinking Clock Bug

A second Clock bug showed up when the sequence in the center of Figure 5 was run. The images at the top of Figure 5 show how the date is supposed to be displayed in MM/DD/YY format. The images at the bottom show how the two-digit year is left off after the center sequence is executed. The bug was originally detected in Analog mode (since the test program cannot read the date in Digital display mode), but the bug was easy to reproduce manually in Digital mode.

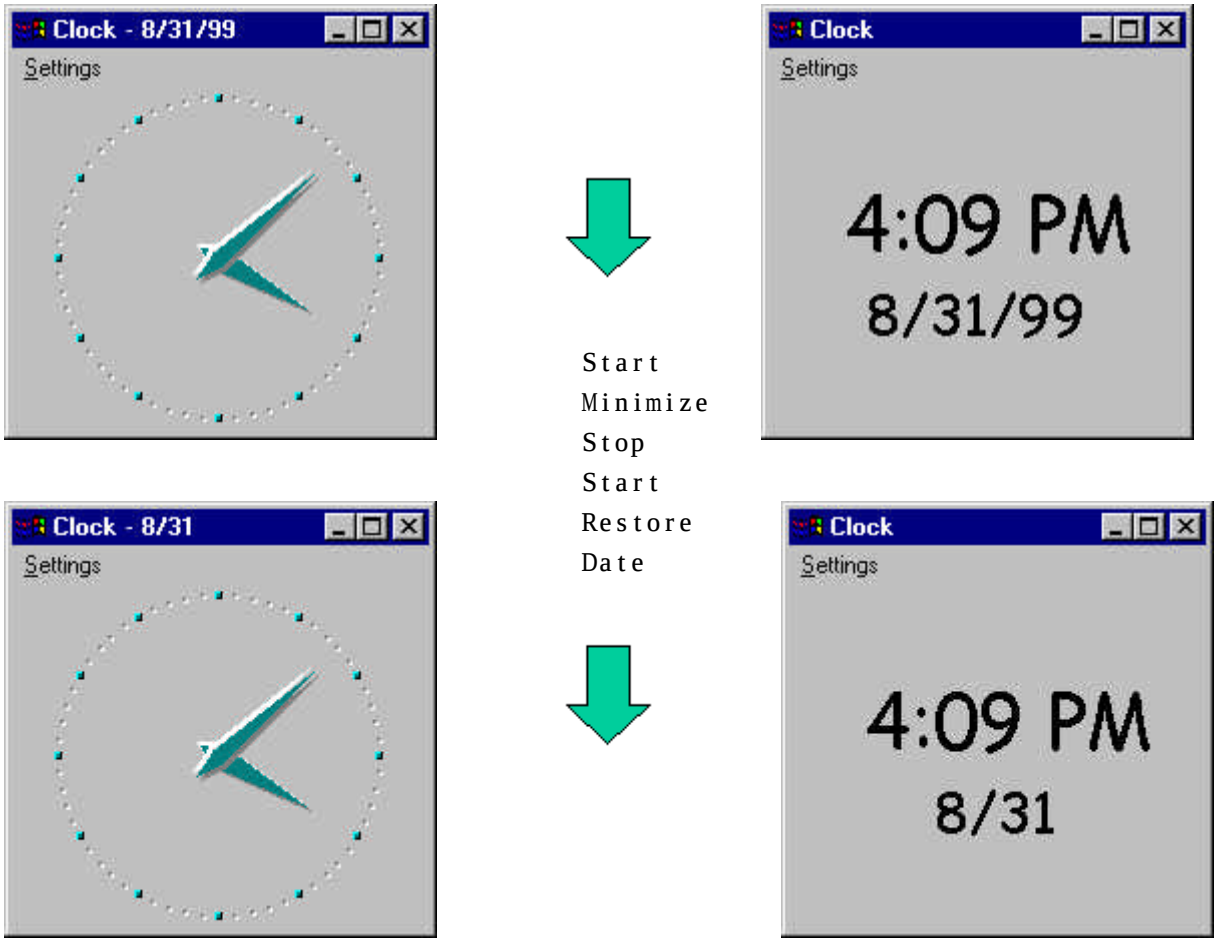


Figure 5: Where Have the Years Gone?

This anomaly in the date was detected by an test oracle routine that looked something like this:

```
title = GetText(0)                                          'get window title

if ( setting_mode = ANALOG) _                               ' if we are in Analog mode
AND WMenuChecked("Settings\Date")  then                     ' AND the date is turned on

        if ("Clock - " + date$ <> title ) then              ' window title should include date
                print "window title is wrong:", title
        endif

else

        if ("Clock" <> title ) then                         ' window title should read "Clock"
                print "window title is wrong:", title
        endif

endif
```

Conclusions

Model-based testing is a new and evolving technique that allows us to automatically generate software tests from explicit descriptions of an application's behavior. Because the tests are generated from a model of the application, we need only update the model to generate new tests when the application changes. This makes model-based tests far easier to maintain, review and update than traditional automated tests.

Low-cost general-purpose test programming languages such as Visual Test's BASIC have sufficient power to allow us to create a finite state model and to generate test paths through that model. Application interface functions (such as WMenuSelect and WMenuChecked) enable the test program to manipulate application controls and to verify the state of those controls.

Testers who are willing and able to create model-based test programs can create flexible, useful tests for the cost of a general-purpose test language tool.

For More Information

For more information on model-based testing, including a fuller discussion of the Clock model and a bibliography of model-based testing papers, please go to the **www.model-based-test.org** website.

References

[1] Visual Test is a trademark of Rational Software Corporation.

[2] Apfelbaum, Larry. "**Model-Based Testing**", **Proceedings of Software Quality Week 1997**

[3] Beizer, Boris. **Black Box Testing: Techniques for Functional Testing of Software and Systems,** New York, John Wiley & Sons, 1995

[4] The Clock application used in this paper is found in Windows NT 4.0 (Build 1381 Service Pack 4).

[5] Whittaker, James A. and El-Far, Ibrahim K. "**Automated Construction of Behavior Models for Software Testing**", **IEEE Transactions on Software Engineering,** (submitted)

[6] Robinson, Harry. "**Graph Theory Techniques in Model-Based Testing**", **Proceedings of the International Conference on Testing Computer Software 1999**

Biography

Harry Robinson is a software test engineer with the Intelligent Search Test Group at Microsoft. He has a BA degree in Religion from Dartmouth College and a BS and MS degree in Electrical Engineering from the Cooper Union for the Advancement of Science and Art.

He was a software developer for six years before coming to his senses and switching to testing. Prior to joining Microsoft in 1998, he spent ten years with AT&T Bell Laboratories and then three years with Hewlett-Packard. He is a long-time advocate and practitioner of model-based testing.

HARRY ROBINSON

Harry Robinson is a software test engineer with the Intelligent Search Test Group at Microsoft. He has a B.A. degree in Religion from Dartmouth College and a B.S. and M.S. degree in electrical engineering from the Cooper Union for the Advancement of Science and Art.

He was a software developer for six years before coming to his senses and switching to testing. Prior to joining Microsoft in 1998, he spent 10 years with AT&T Bell Laboratories and then three years with Hewlett-Packard. He is a long-time advocate and practitioner of model-based testing.