State-based testing using dynamic instrumentation – A Case Study

Author
Vijay Upadya, vijayu@microsoft.com

Abstract

Often times while testing state-based systems, testers have the need to validate the behavior of software on performing a specific action while a component is in a specific state. However this is not easily possible using traditional testing techniques because the state transitions are either not visible/observable externally to tests or, in cases where they are visible, are not controllable in real time by tests. One technique that addresses this challenge is dynamic instrumentation, in which the system under test is dynamically instrumented using binary interception techniques that then enable tests to observe and control the product as it transitions through its various states.

Testing state-based systems can be quite challenging and costly. The primary challenge in testing such systems is the need for deterministically covering all the states and state transitions of interest in an effective and efficient manner. This paper will talk about the technique we are using in our team to test such a system using dynamic instrumentation. The paper will discuss this testing approach, the tools used to do dynamic instrumentation and how to write tests to leverage the dynamically instrumented product code to simulate different states and state transitions. This paper will also demonstrate how the approach lends itself well for creating data driven tests and ways it can be used to augment the existing test automation.

Biography

Vijay Upadya is a senior software design engineer in test at Microsoft's Redmond campus in Washington. He has 14 years of industry experience and is currently working in the Microsoft Windows Live group and primarily focuses on test strategy, test tools development and test process improvements for the team.

Vijay Upadya has Master's degree in Systems Analysis and Computer Applications from the University of Mangalore, India and Bachelor's degree in Mechanical Engineering from Gulbarga University, India.

1 Introduction

Testers testing state-based systems have a need to validate the behavior of the system under test (SUT) on performing a specific action while the product is in a specific state. However such testing is not easily possible using traditional test approach because:

- The state transitions of the SUT are either not visible/observable externally or in cases where they are visible, they are not controllable in real time.
- Traditional functional tests run out-of-process with the system under test.

Such was the situation when our team took on testing file synchronization software (Wikipedia 2012a) named *SkyDrive* (Wikipedia 2012b). This software is a client application that synchronizes files across users' devices as well as synchronizes files to their stored 'cloud' locations. Although the client application is a state-based system, not all of its states and transitions are externally visible to the end user. The behavior of the application varies depending on its state, which is changed through user actions. For example, the code path followed by the application's execution is different while the user is editing a file than the code path followed while that same file is being uploaded. And the code path is yet further different when the user is editing a file while the application is in an idle state. Trying to test these states and transitions posed a great challenge to the team.

2 Approach

The key to being able to test different conditions was to be able to observe and control the state transitions in the product. The solution the team used to achieve this level of control was *dynamic instrumentation* of the SUT. In this approach, key functions in the product that cause state transitions were identified. Instrumentation code was then added to notify the test program when different state transitions happened. The test program then performed appropriate steps to trigger the state transitions and was able to validate the SUT behavior deterministically (i.e., in a repeatable fashion).

The framework that we used for this dynamic instrumentation was *detours*. *Detours* (Galen, 1999) is a library for instrumenting arbitrary Win32 functions in Windows-compatible processors. The *detours* framework provides methods to intercept Win32 functions by re-writing the in-memory code for target functions. The library also contains utilities to attach arbitrary DLLs and data segments (called payloads) to any Win32 binary. One of the big advantages of this approach is that since the interception happens at the binary level dynamically, instrumentation doesn't require any change to the product source code. *Detours* express library can be freely downloaded from the link given in the reference section at the end of this paper (Microsoft Research, 2012).

We used *detours* functionality to implement a test library, which could then be used to observe and control the SUT in real time as it moved through its various states. In its supported usage, a test library implemented with *detours* can be injected (i.e., loaded) into the application process dynamically at run time while executing the tests.

3 Implementation

3.1 Basic setup

- The test program (Test.exe) used *DLL injection* (Wikipedia 2012c), which is a technique used to run code within the address space of another process by forcing it to load a dynamic-link library (i.e., a DLL). The *detours* test DLL (detourTest.dll) is injected into the product process (Product.exe) explicitly by the test and detour methods that are appropriate for the test are enabled.
- The *detours* test DLL *detours* necessary product functions (could be public or private). Since the *detours* DLL runs in-process with the product process, it has full access to the internal state of the product.

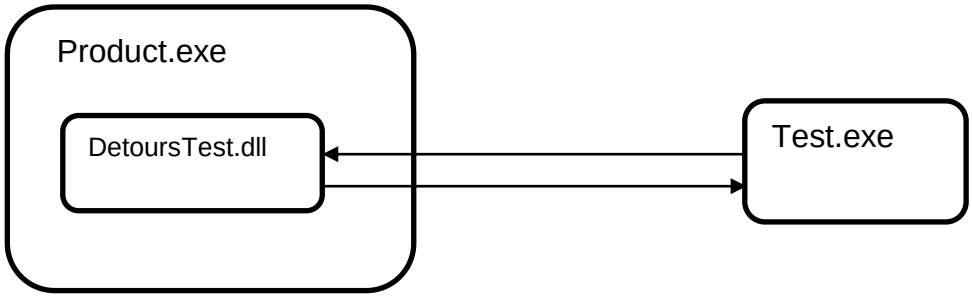

Figure 1: Test execution using detours

In the above diagram, the Test.exe launches the system under test (Product.exe) and loads the DetoursTest.dll into the product process explicitly. Then it sends commands to the SUT to trigger various state transitions and waits for event notification back from DetoursTest.dll.

3.2 Test Steps

- The *detours* component fires events for all product state transitions that are instrumented for a given scenario
- The test program records various events fired for that scenario
- The test program then inspects these recorded events, repeats the scenario and performs appropriate actions while product is in a specific state. For example, test.exe can perform a file edit while that file is being uploaded.
- The *detours* DLL also enable tests to 'block' the product when it reaches a specific state to give the test program an opportunity to perform its actions deterministically (else by the time an action (like edit file) is performed, product might leave that state and transition to a different state).

All of the above steps are automated and don't require any user interaction.

The following example will demonstrate this sequence more clearly:

3.3 Example

Let's take an example test case of editing a file while the product is uploading it.

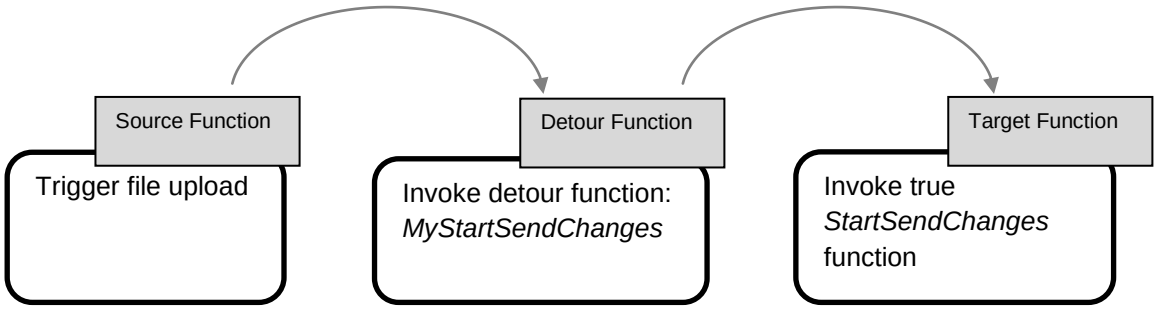

Figure 2: Detours execution flow

Step1: Identify the method in the product that performs file upload. Say this is the signature of that method -

```
void StartSendChanges(Changeset* changes)
```

Step2: Create a test wrapper in DetoursTest.dll for the product function 'StartSendChanges', which overrides the product code by using the *detours* method - DetoursAttach. When the product code calls 'StartSendChanges', the actual function being executed will be 'MyStartSendChanges', allowing Test.exe to record the product state."

```
void MyStartSendChanges(Changeset* changes)
{
	SignalAndWait(ProductStates::UploadBegin);

	//Call real product function
	StartSendChanges(changes);
}

//Attach the detour method
DetourAttach(StartSendChanges, MyStartSendChanges);
```

DetourAttach is a helper method that is part of detours library that redirects the product's function pointer to wrapper functions. *SignalAndWait* is a helper function called from inside the detour wrapper method *MyStartSendChanges* shown above. This function fires '*UploadBegin'* event to notify listeners (test code) that the product is in the 'Upload' state and waits until it is signaled by the test. Once signaled, the original product function *StartSendChanges* is called to resume the execution flow. Figure 2 above shows the execution flow.

Step3: In the test following steps are performed

- Start product
- Inject *detoursTest.*dll to product.exe
- Trigger file upload
- Block the product when it goes to 'upload' state
- Perform file edit. At this point, the product is guaranteed to be in 'upload' state
- Resume upload
- Verify that upload resumed and completed successfully

4 Data driven tests

Once the basic flow of the test was identified and implemented, the next task was to prepare the list of all possible state transitions that needed to be tested and actions that needed to be performed in each of those states. The test team identified about 30 states (e.g., file upload begin state, file upload complete state, etc.) and 15 user actions (e.g., file edit, file delete, etc.) that needed to be tested. In order to test all these cases efficiently, we built a simple data driven test (Wikipedia 2012d) framework that took a list of actions and states in the form of an xml document and generated test cases by combining these two pieces of data.

For example, let us say we have to test four actions-: edit, delete, move and rename of files while the product is a) uploading and b) downloading the file. This information is specified in an xml file as shown below:

```
<TestData>
  <Actions>
    <EditFile/>
    <RenameFile/>
    <MoveFile/>
    <DeleteFile/>
  </Actions>
  <States>
    <Uploading/>
    <Downloading/>
  </States>
</TestData>
```

The data driven test framework parsed the above xml data and generated 8 test cases (4 actions x 2 states). Each of the test cases was then executed as explained in the above section 3.2. This data driven test approach enabled the test team to rapidly test in an efficient manner.

5 Coverage and Cost

Before we adopted the state-based testing approach, more than 60% of states in the product were just not possible to test deterministically because there was no way for the test program to observe and control these state transitions. The test team had no test coverage for these areas. This state-based testing approach enabled the test team to rapidly test these missed states and get the much needed test coverage.

In terms of the investment, it took 15 person days to get the *detours* based test infrastructure up and running. Then it was just a matter of incurring incremental cost for defining new states and actions and enabling them in the *detours* test library. Due to the data driven nature of test definition and execution, this incremental cost is much less than if the same tests were to be written in a non-data driven way. For example, a typical test would take approximately half a day to one day to implement, whereas a data driven test would typically take about an hour to implement.

6 Other Applications of Dynamic Instrumentation

While the above section shows how *detours* can be leveraged to do targeted functional test, there are other applications of dynamic instrumentation. Below are some of the areas that can be applied:

- **Deterministic/Controlled stress testing:** Perform various real world operations (edit file, delete file, network off, etc.) while file synchronization is progressing through various states.
- **Crash resiliency testing:** Crash client while it is in a specific state and test its resiliency (restart and sync should continue from where it left before crash)
- **Delay/Timing resiliency testing:** Run existing functional tests by introducing random delays in various stages of syncing process. This is to catch any timing related issues in the product.
- **Error testing:** Modify return value of function calls to simulate error conditions. This is a cheap way to test error conditions that typically tend to be expensive/hard.

7 Conclusion

Dynamic instrumentation using *detours* enabled testing of state-based systems effectively and improved the coverage of testing different states. This methodology does require initial investment to identify different states and instrument the product code based on these states, but once it is in place, it can rapidly enable test teams to test different conditions/code paths in the product in a deterministic manner.

Using this framework, and our data drive test methods we could cover 60% of additional state transitions that were not possible to test before and add new tests in an hour instead of a day.

8 References

Wikipedia 2012a, File Synchronization, http://en.wikipedia.org/wiki/File_synchronization (accessed August 16, 2012)

Wikipedia 2012b, SkyDrive, http://en.wikipedia.org/wiki/SkyDrive (accessed August 16, 2012)

Wikipedia 2012c, DLL Injection, http://en.wikipedia.org/wiki/DLL_injection (accessed August 16, 2012)

Wikipedia 2012d, Data-driven testing, http://en.wikipedia.org/wiki/Data-driven_testing (accessed August 16, 2012)

Hunt, Galen and Brubacher, Doug. 1999. *"Detours: Binary Interception of Win32 Functions".* *http://research.microsoft.com/apps/pubs/default.aspx?id=68568*

Detours, Microsoft Research, 2012. http://research.microsoft.com/en-us/projects/detours/