State-based Testing Using Dynamic Instrumentation

Vijay Upadya
Microsoft

Agenda

* What is dynamic instrumentation?
* Applications of dynamic instrumentation
* How does it work?
* Simulating different states and transitions deterministically
* Creating data driven tests
* Results
* Conclusion

Introduction

* System Under Test : SkyDrive Sync application
  * Synchronizes users files/folders across devices and cloud
  * Attributes of this app
    * State-based system
    * State transitions not visible externally to user (only end results are)
    * State transitions not controllable externally

Example Scenario

* Scenario: User edits a file locally
* Verify: Edit propagates to other devices and cloud

* Code path is different based on whether
  - Product is idle
  - Product is in the middle of uploading that file
  - Process is in the middle of downloading changes to that file
  - Product is in the middle of processing that file (e.g. scanning, hashing)

What is Dynamic Instrumentation?

* Technique where instrumentation is performed at run time (in–memory, not on disk) on the compiled binary files

* Does not require recompilation of source code after adding instrumentation

Why use Dynamic Instrumentation?

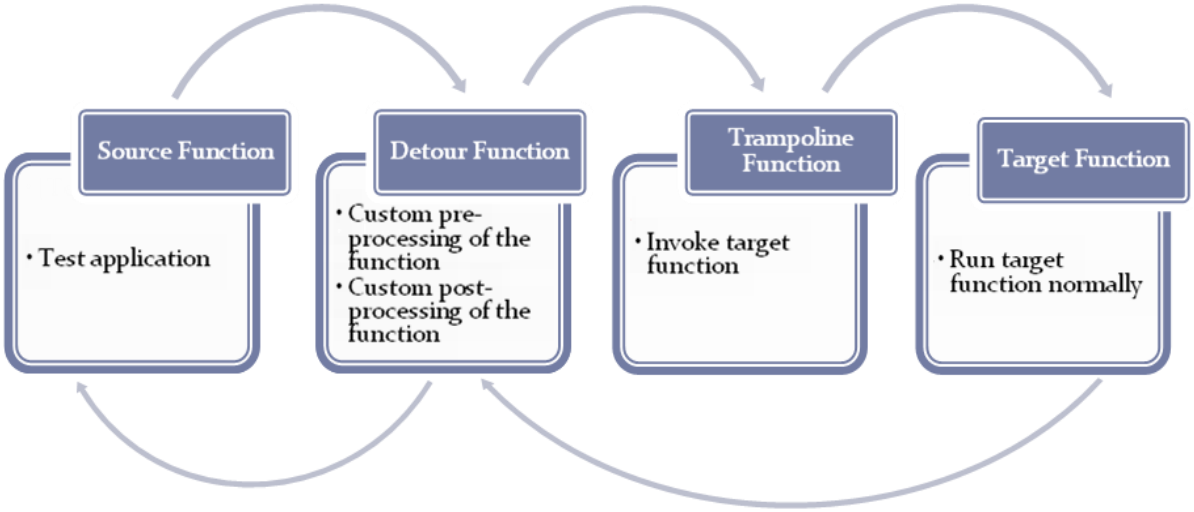
* Imagine wanting to test something that you don't necessarily have direct testing access
* Imagine having the ability to be the "man in the middle" and being able to intercept data that you could otherwise not get
* Imagine wanting to trace data though a scenario and validating every step instead of just the end result

What is detours?

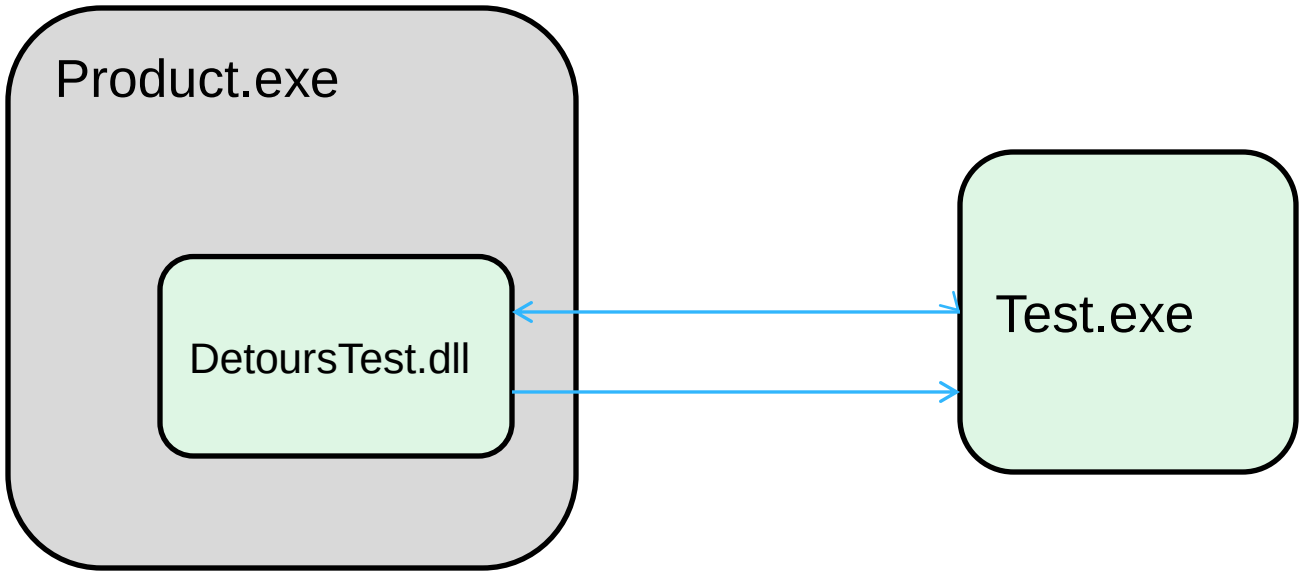
* Library toolset developed by MS Research for intercepting function calls
* Interception applied dynamically at runtime (in–memory, not on disk)
* Enables either replacing the target function or extend its semantics

How Detours Works

The Overview



Detours- Implementation



How to use detours?

* Test library implemented using detours library
  * Intercepts product functions that cause state transitions
  * Fires event for each state transition (i.e. *observe* transitions)
  * Enables pause/resume of execution flow (i.e. *control* transitions)

* Test injects detours test library into product's process to observe states and control transitions

Detour Test Steps

* Perform user action
* Observe – all states transitioned
* Redo the same user action
* Control - when interested state is reached
  * Pause product execution
  * Perform test action
  * Resume product execution

Example

* Test scenario: Edit file while its being uploaded

Step1:

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

```
void StartSendChanges(ChangeSet* changes)
```

Example (cont.)

Step2:

Detour ‘StartSendChanges’ function in test library

```
//Implemented in detourstest.dll

void MyStartSendChanges(ChangeSet* changes){

  SignalAndWait(State::UploadBegin);

  //Call real product’s function
  StartSendChanges(changes);
}
```

Example (Cont.)

Step3:

Attach detour method by calling DetourAttach API

```
DetourAttach(StartSendChanges, MyStartSendChanges);
```

Example (Cont.)

Step4:

Execute test that performs following -

* Start product
* Inject *detoursTest.*dll to product.exe
* Trigger file upload (e.g. create a new file)
* Block the product when it goes to ‘upload’ state (indicated by event fired by detourstest.dll)
* Perform file edit. At this point, the product is guaranteed to be in ‘upload’ state
* Resume upload
* Verify that upload resumed and completes successfully

Data Driven Tests

* 30 states identified to be tested (upload, download, scanning, etc.)
* 15 user actions (edit, delete, rename, etc.)

* Challenge: How to write test to cover these efficiently?

Data Driven Test Example

4 actions - Edit, delete, move and rename of a file while the product is in 2 states - Uploading , downloading

Total = 8 test scenarios (4 actions X 2 states)

Test xml

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

Test Coverage and Cost

* 60% more states and state transitions covered
* 15 person days to develop infrastructure
* Incremental cost to add more scenarios
  * Went from a day to an hour to enable new state transition scenario

Other Applications of Dynamic Instrumentation

* **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.

Conclusion

* Initial upfront investment for infrastructure
* Increased state coverage
* Decreased cost for adding new scenario
* Opens up new avenues of testing

Questions?