Deterministic Test Automation of client application in a Client-Server Environment

Prashanth Uppunda (Prashanth.Uppunda@siemens.com)
Niladri Dey (Niladri.Dey@siemens.com)
Sneha Joseph (Sneha.Joseph@siemens.com)

Siemens Technology & Services Private Ltd.
CTDC AA I,
Salarpuria Infozone,
Electronics city, Hosur Road,
Bangalore - 560100
Abstract

It is a well known fact that, test automation helps in reduced efforts for testing of software systems, as it avoids the manual efforts required for regression testing. However, there are scenarios or situations in which it is considered to be either not feasible or too difficult to automate. One such scenario is to deterministically test a client program in a client-server environment where multiple client actions lead to concurrent scenarios.

Deterministic automation testing of concurrent scenarios of a client program in client-server environment are often considered complex because of concurrent server access from several clients running parallely. Hence, these scenarios are typically limited to only manual testing. Deterministic testing for scenarios such as simultaneous server calls from multiple clients is almost impossible to test manually too.

In our endeavor to find a solution for deterministic automated testing of concurrent scenarios of a client program in client-server scenario, we have developed a unique and innovative solution which is the core theme of this paper. Here deterministic means the test results are always consistent and it is always possible to predict the output of an automated concurrent scenario test.

Problem Scenario

In a software system where multiple clients can connect to a common server and perform parallel execution of operations, it is often complex to adapt a test strategy to deterministically automate concurrent execution scenarios.

Parallel execution of operations in a test environment to check for possible issues related to resource access by concurrent client actions is very important to ensure the quality of client programs. If the client program is not developed properly, typically with improper error/exception handling, it will lead to undesired situations during execution of such client programs. In addition, it might not be possible to think of all concurrent access scenarios from various clients at the beginning of development, leading to some scenarios left untested at the unit level. Furthermore automation of such test scenarios is very important to maintain the quality of software in a changing environment (regression tests)

Following figure (Figure 1) shows a typical client server environment, with several clients acting on the server to perform various actions concurrently.
Following are some potential example scenarios where it might become difficult to deterministically test the outcome or find issues.

1. One client is creating a resource under a specific parent node. The program execution is at a point, where we have obtained the handle of the parent node and about to add the new resource under this parent. At this instance, another client tries to delete the parent node, and it gets deleted by this action. Now, if proper error handling is not done at the client side, execution triggered by the first client might lead to undesired behavior such as crash of the client program. To add to the complexity, such situations are almost impossible to test until unless the concerned developer puts efforts in simultaneous debugging of multiple clients, by putting breakpoint at one instance and executing the other instance.

2. Two clients are trying to delete a common resource at the same time. In an ideal scenario, the resource should get deleted by action of one of the client, and other client action should lead to an error. However, if the error handling is not proper, this too might lead to undesired behavior such as crash of the client program.

Considering several such actions from multiple clients running parallely, it becomes even more difficult to test for combinations of scenarios.
**Existing approaches**

As of our knowledge and searches on the web, at this moment there are following approaches to address this problem.

Manual testing by executing related test cases almost at parallel and checking for any errors / expected results is the most common approach. Another manual approach at the unit testing level is to debug the program in two different developer studio environments (say two different Visual Studio instances), by placing breakpoints at relevant location of one instance, and then executing other instance to see an expected results / errors.

Mocking frameworks such as Rhino-Mocks can be used to ‘mock’ the server behavior and tests can be written to ‘expect’ an error in a certain scenario. However, mocking has limitations to test at the ‘unit test’ level only and not at the real server environment. Also, building a server mock has limitations, as any change in new versions of server interface will go undetermined until they are actually discovered during manual testing.

There are other readily available tools in the market such as Rational Performance Tester, which only help to simulate concurrent scenarios to assist in load/performance testing scenarios and not deterministic testing of parallel execution scenarios.

As of our knowledge there is no test automation concept to deterministically test a client program for concurrent multi-client scenarios in a client-server system.
Solution Description

In a client server environment, a client operation comprises of executing several server calls. For the purpose of explanation of our idea, we call this exact code location where there is a server call as wait-point.

For e.g.:

If we consider creation of a resource under a parent node in server as a use case, it would have following possible wait points
1. Get parent node handle
2. Create first part of new resource
3. Create second part of the new resource
4. Associate second part with first part
5. Associate the new resource with parent node

Similarly deletion of a resource under a parent as a use case might have following wait points
1. Get parent node handle
2. Get handle of the resource to be deleted
3. Disassociate second & first part of the resource to be deleted
4. Disassociate first part from its parent node
5. Delete second part
6. Delete first part

Our strategy is to make execution of one instance of the client component wait at a specific wait point during a specific client operation, and execute another related operation from another instance of client component, and observe the expected behavior/results.

The following steps better explain our approach

(1). A client-server parallel test support library (MutLib) is developed with following functionalities
This library makes use of operating system objects such as an event of Windows, to make the program execution wait/fallback at a specific wait point.

(2) In the component under test (client component which interacts with server), wait points are designated by special encoding in following format

`//WAIT_POINT:Create_Resource_Before_Associate_New_Resource_With_Parent`

These are commented lines which component developer prescribes during development, to designate a wait point.

(3). A parser and generator module will parse the source code of this component and replace all encoded (commented) wait points to function calls. That means the above wait point comment will be converted in to the following function call

```csharp
public void SetWaitPoint(string waitPointId)
Method which will be used by Test Automation code to 'set' a specific wait point, so that the execution 'waits' at that specific location for that instance of component
```

```csharp
private void SetWaitPoint(string waitPointId, DateTime waitUntil)
Method which will be used by Test Automation code to 'set' a specific wait point, so that the execution 'waits' at that specific location for that instance of component until specified time.
```

```csharp
public void ResetWaitPoint(string waitPointId)
Method which will be used by Test Automation code to reset waiting at a specific wait point. The execution falls back once this is done.
```

```csharp
public void WaitAtWaitPoint(string waitPointId)
Method which will be used by component instance to wait at a certain 'waitpoint' identified by a unique name.
```

Furthermore, the parser and generator module generates the new method “WaitAtWaitPointInternal”, which take care of dynamically loading the “MutLib” library and calling the `WaitAtWaitPointInternal()` method of the library.

Further, it is made possible to turn off calling the `WaitAtWaitPoint()` library method via a special property named “MutMode” defined on the component under test. That means, if MutMode=true, then only the `WaitAtWaitPoint()` method on MutLib will be called.

This call makes sure that, if the wait point named "Create_Resource_Before_Associate_New_Resource_With_Parent" is set by the test automation code, and then the component is executed as part of the client program, the program execution will halt at this location.

(4). The component whose source code is modified by way of step (3) is then compiled and the updated component is generated. This generated component will be used for concurrent multi client automated tests.

(5). Test automation code is written using the MutLib and referencing the component under consideration and then the tests are executed.
The below block diagram (Figure 2) summarizes the steps explained above.

Figure 2: Solution overview
**Key Execution Scenarios**

Some of the key scenarios we automated using this approach are depicted below.

1. Waiting at wait point while executing a scenario from one thread (first client) and executing a different scenario from another thread (second client).
(2). Both the threads (clients) waiting at same wait point for same scenario, and falling back at the same time to execute operation parallely.

- **Thread 1**
  - MutMode = true
  - (1) Set MutMode = true
  - (2) Connect("User1")
  - (3) Connect("User2") (Seperate Thread)
  - (4) SetWaitPoint("Create_Resource_Before_Associate_New_Resource_With_Parent", 10:15:00)
  - (5) CreateArtifact("Folder1\Screen_1")
  - (6) WaitAtWaitPoint("Create_Resource_Before_Associate_New_Resource_With_Parent")
    - Wait at "Create_Resource_Before..."
  - (8) SetWaitPoint("Create_Resource_Before_Associate_New_Resource_With_Parent", 10:15:00)
  - (9) CreateArtifact("Folder1\Screen_1")
  - WaitAtWaitPoint("Create_Resource_Before...")
    - Wait at "Create_Resource_Before..."
  - (14) Return result (exception?)
    - Finish wait at "Create_Resource_Before..."
  - (15) Return result (exception?)
    - Finish wait at "Create_Resource_Before...
  - (16) Check for expected result (Exception in one of the Component instance)

- **Thread 2**
  - MutMode = true
  - (1) Set MutMode = true
  - (2) Connect("User1")
  - (3) Connect("User2") (Seperate Thread)
  - (4) SetWaitPoint("Create_Resource_Before_Associate_New_Resource_With_Parent", 10:15:00)
  - (5) CreateArtifact("Folder1\Screen_1")
  - (6) WaitAtWaitPoint("Create_Resource_Before_Associate_New_Resource_With_Parent")
    - Wait at "Create_Resource_Before..."
  - (8) SetWaitPoint("Create_Resource_Before_Associate_New_Resource_With_Parent", 10:15:00)
  - (9) CreateArtifact("Folder1\Screen_1")
  - WaitAtWaitPoint("Create_Resource_Before...")
    - Wait at "Create_Resource_Before..."
  - (14) Return result (exception?)
    - Finish wait at "Create_Resource_Before..."
  - (15) Return result (exception?)
    - Finish wait at "Create_Resource_Before...
  - (16) Check for expected result (Exception in one of the Component instance)
**Outcome**

The key advantages of this approach is the ability to automate concurrent execution scenarios in a client program in client-server environment which would not only save huge efforts on manual testing of such scenarios, but also makes it possible to test for scenarios which are almost impossible to test using manual ways of testing. Replacing manual tests of such scenarios using debug/breakpoint approach with in-house developed automated approach reduces the regression test efforts by 100%. Scenarios which are complex or impossible to test manually are automated using this approach.

Since the testing is performed on the real server environment and not a mocked or simulated environment, any changes to the server interface or server behavior (for e.g. due to a new version of server) will immediately be discovered due to a regression test failure. For e.g. a new version of the server interface method throws a new exception which is a change implemented to its behavior from previous version. However, due to some reasons, the client developers did not modify their client program accordingly, and so the client programs would exhibit undesired behavior during runtime. Such situations are immediately discovered during regression runs using our approach.

This concept has been successfully implemented in one of our key project, by automating 200+ client server parallel test scenarios. This enabled us to unearth several issues caused by improper or missing error handling in client side code.

**Inventiveness & Reuse**

The inventiveness of this approach are – a test automation strategy where a client program execution is made to halt at a specific location (wait point) during runtime, and another related action is performed at the same time from a different client connected to same server, to see the expected behavior of the client program.

Although the desired purpose of our MutLib is to support us in above explained concurrent client scenario testing, it can be re-used to any purpose which requires a program to pause/continue in a controlled passion during runtime.
References

[1]. Rational Performance Tester.  
[2]. Intel Thread Checker  
[3]. Cheer-Sun D. Yang and Lori L Pollock. The challenges in testing of Multithreaded programs  
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.8791&rep=rep1&type=pdf  
[4]. Jakob Nielsen’s. Multiple-User Simultaneous Testing (MUST)  
http://www.nngroup.com/articles/multiple-user-simultaneous-testing/  
[5]. Microsoft Research. CHESS: Find and Reproduce Heisenbugs in Concurrent Programs  
[6]. Pentti Pohjolainen. SOFTWARE TESTING TOOLS  
[7]. Ayla Dantas, Francisco Brasileiro, Walfredo Cirne. Improving Automated Testing of Multi-threaded Software  

Authors Biography

Prashanth Uppunda
Prashanth Uppunda has over 16 Years of experience in software development and is currently working as Senior Software Architect in Corporate Technology Development Center - Bangalore (CTDC), a division of SIEMENS AG, Germany. He has worked on various products related to industrial automation software solutions. He is a SIEMENS certified Software Architect (SWA), and his current responsibilities include requirement analysis, defining the software architecture including test design techniques and solution design.

Niladri Dey
Niladri Dey has over 6 years of experience in software test areas and is currently working as Senior Test Engineer in Corporate Technology Development Center - Bangalore (CTDC), a division of SIEMENS AG, Germany. He is an ISTQB certified test engineer and his current responsibilities include designing test automation solutions and development of test suites.

Sneha Joseph
Sneha Joseph has over 2 years of experience in software test areas and is currently working as Test Engineer in Corporate Technology Development Center - Bangalore (CTDC), a division of SIEMENS AG, Germany. Her current responsibilities include designing & developing test automation solutions and manual testing.
## Appendix

### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MutLib</td>
<td>Multi User Test Library. The library developed to facilitate automation of client-server scenarios</td>
</tr>
<tr>
<td>wait-point</td>
<td>A designated location in source code where we wish to wait during test execution. Typically locations where server calls are made</td>
</tr>
<tr>
<td>Mocking framework</td>
<td>Frameworks used to facilitate unit testing by simulating the dependent environment</td>
</tr>
<tr>
<td>Rhino-mocks</td>
<td>A specific mocking framework</td>
</tr>
<tr>
<td>Library</td>
<td>Helper component or a .NET assembly</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Designated locations of source code used to halt program execution during debugging</td>
</tr>
</tbody>
</table>