Test Coordination and Dynamic Test Oracles for Testing Concurrent Systems

Bernard Stepien, Liam Peyton
School of Engineering and Computer Science
University of Ottawa
Ottawa, Canada
Email: {bstepien | lpeyton}@uottawa.ca

***Abstract*—Testing concurrent systems is complex. In traditional software unit testing, a test sequence is always composed of a stimulus and its corresponding fully predictable response. With concurrent systems, this simple model no longer holds as the state of the System Under Test (SUT) changes while several users place their requests. Race conditions are a particularly challenging problem for testing, since they will occur and must be identified, but are very disruptive to the test environment. In this paper, a case study, using the formal test specification language TTCN-3, illustrates the challenges for test coordination, especially race conditions, and propose techniques to address them. We also introduce shared variables and the use of semaphores in the TTCN-3 parallel test component model as a mechanism to implement dynamic test oracles.**

Keywords- software testing-concurrent systems; TTCN; test oracles; race conditions.

I. INTRODUCTION

Testing concurrent systems is complex. In traditional software unit testing, a test sequence is always composed of a stimulus and its corresponding fully predictable response [4]. With concurrent systems, this simple model no longer holds as the state of the system under test (SUT) changes while several users place their requests. Race conditions are a particularly challenging problem for testing, since they will occur and must be identified, but are very disruptive to the test environment.

Some definitions and implementations of parallel testing can be found in [6][7][8][9]. Obviously there are different kinds of parallel testing. In the previous reference, the main concern is to run sequential tests in parallel in order to save time. Instead, we focus on concurrent testing of states in a system under test (SUT) states as the test purpose. There are two main categories of concurrent test systems:

- Response time testing when a large number of requests are sent to a server as shown in Figure 1. This is addressed using TTCN-3 in [10].
- Testing the actual processing logic of the SUT when confronted by several requests from parallel users where the state of the SUT is changing as a result of requests of the users and thus affecting each user's behavior.

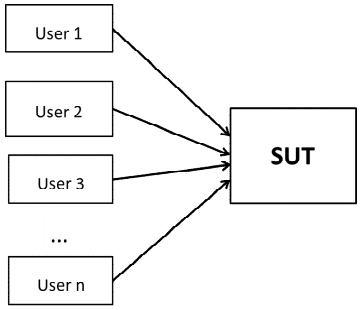

Figure 1. Parallel system configuration

In this paper, a case study, using the formal test specification language TTCN-3, illustrates the challenges for test coordination, especially race conditions, and proposes techniques to address them. We also propose shared variables and semaphores in the TTCN-3 parallel test component model as a mechanism to implement dynamic test oracles. Overall, the motivation to use a formal method such as TTCN-3 and its related available execution tools is to take full advantage of its logging information in order to rapidly detect faults due to race conditions. We also propose enhancements to the TTCN-3 language to make our testing concurrency problem statement usable.

II. A CASE STUDY

In sub-section A we define the dynamic state problem to be addressed, in sub-section B we propose thee methods to specify concurrent systems tests.

A. Defining the problem

Although, we have studied extensively testing concurrency problem in industrial applications [11], the following simplified case study is about testing the transition of the state of a system and the kind of responses it should reply with. Here we have parallel users that send a request to a book ordering system and get two kinds of replies depending on the two possible states of the SUT: has stock; or out of stock. The problem is that it is impossible to predict the test oracle (predicted response) since each user is independent from each other and thus does not know the state of the SUT. This is similar in e-commerce applications like on-line ordering of merchandise and hotel booking and train or airline reservations systems. A typical warning message for a hotel reservation system is to warn the customer that there

 ISBN: 978-1-61208-776-4

is only one room left at a given rate. Thus from a tester point of view, it is hard to predict if a response corresponds to a success or a failure. However, if the users are coordinated, the response to a given user can be predictable.

The interesting aspect of this simple example is that we have tried various approaches of coordination and some resulted in race conditions problems, thus disturbing the test process altogether. Table I shows the values of test oracles depending of the state of the SUT, in our case: has stock; or out of stock. In short a test passes if an invoice and shipping confirmation is received when there is inventory left or when out-of-stock is received and the server is out of stock. All other cases are failures.

Unit testing would consists in putting the SUT in the appropriate state and check the individual responses.

What is missing from a unit test is the dynamic aspect of seeing the state change as the maximum available inventory is reached.

TABLE I. EXPECTED TEST ORACLES DEPENDING ON THE STATE OF THE SUT

Response to the User/state	Has stock	Out of stock
Invoice	pass	fail
Out of stock	fail	pass

B. TTCN-3 implementation

The TTCN-3 implementation of the user parallel test component (PTC) is based on a simple request/response behavior pattern with the response being analyzed with the four possible configurations of two states and two corresponding responses making use of the TTCN-3 *alt* (alternative) construct. Each alternative is guarded with the predicted state of the SUT. The receive statement contains what the received message from the SUT should match and the predicate between square brackets, the predicted state of the SUT.

```
function ptcBehavior() runs on PTCType
{
  p.send("purchase");

  alt {
     [state == "has_stock"]
        p.receive("invoice") {
            setverdict(pass);
        }
      [state == "out_of_stock"]
        p.receive("invoice") {
            setverdict(fail);
        }
      [state == "out_of_stock"]
         p.receive("out_of_stock") {
            setverdict(pass);
         }
      [state == "has_stock"]
         p.receive("out_of_stock") {
            setverdict(fail);
         }
  };
}
```

Figure 2. PTC Client test verdicts situations

Instead, unit testing would break down the problem into two separate test cases and especially without the need for PTCs. Here the unit is represented by a given state.

First unit test case:

```
function unitTestBehavior_1() runs on
                              MTCType {
  p.send("purchase");

  alt {
     [] p.receive("invoice") {
            setverdict(pass);
        }
     [] p.receive("invoice") {
            setverdict(fail);
        }
```

Second unit test case:

```
     [] p.receive("out_of_stock") {
            setverdict(pass);
     }
     [] p.receive("out_of_stock") {
            setverdict(fail);
     }
   }
```

The predicates are empty because the state is predictable due to the manipulation of the SUT by the tester by emptying the data base in the first case and populating the database in the second case. Another drawback of unit testing is that the testing process would not be entirely automated since it requires a manual intervention of the tester between the two states.

Assuming that the SUT has three books on hand, the ideal testing results would be to get an invoice response for the first three users and an out of stock response for the remaining users as shown on Figure 3 and an overall pass verdict for the test.

However, the results shown in Figure 3 are only ideal and rarely happen. Instead, we see more results of the kind of Figure 4 that show the full effect of race conditions because each PTC starts at different times.

 ISBN: 978-1-61208-776-4

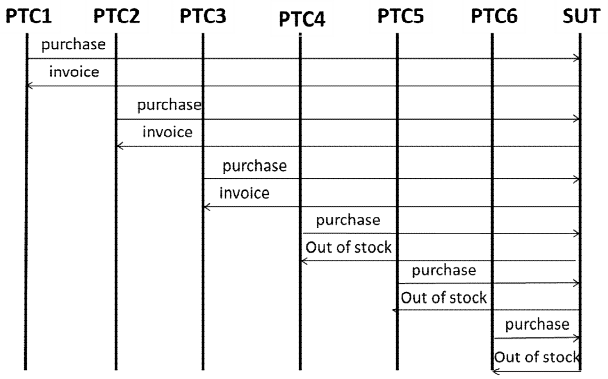

Figure 3. Ideal testing responses

The failures shown in Figure 4 are the result of mismatches between expected and received messages when tests are executed without coordination.

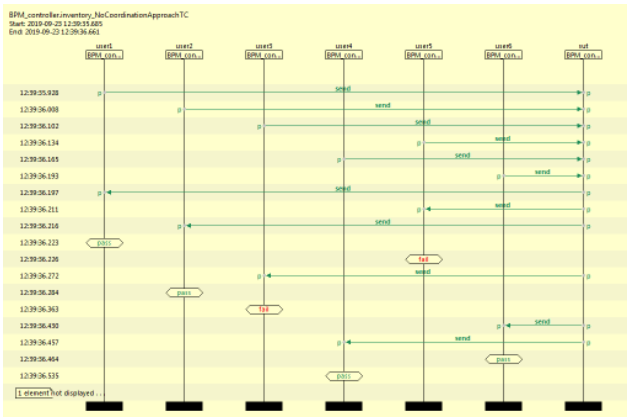

Figure 4. Uncoordinated execution results

Figure 5 shows the TTCN-3 tools data inspection feature [2][3] that provides detailed message and test oracle contents that enable the tester to understand the reasons for failure.

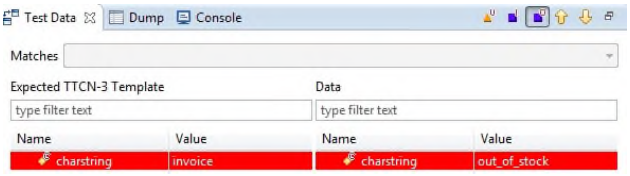

Figure 5. Expected vs received values

In this case, one may wonder where the state value comes from. This is where the test coordination is taking place. TTCN-3 has the concept of main test component (MTC) that precisely looks after that.

In our case the coordination is achieved via abstract coordination ports *cp* that link the master test component and the PTCs as shown in Figure 6.

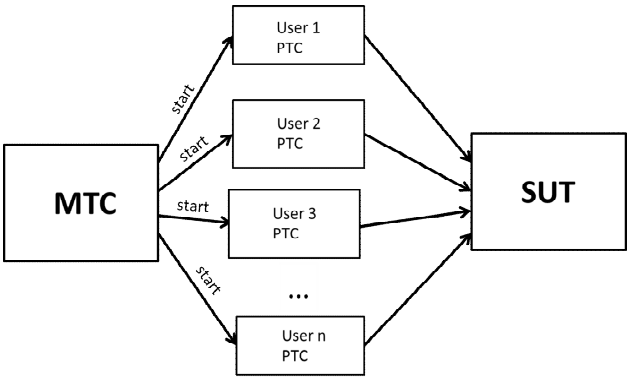

Figure 6. Test coordination with MTC

There are three ways to address test coordination.

1) Using coordination messages

The approach consists in using coordination messages between the MTC and the PTCs that contain the predicted state of the SUT. On the user PTC's side we need an additional line that receives the state from the MTC before the user attempts to test the SUT:

```
cp.receive(charstring:?)->value state;
```

On the MTC side, we send a message containing the state to the PTC that the tester thinks that the server is supposed to be in. In our case this is achieved by changing the state once three requests have been placed as follows:

```
testcase coordinated_msgs_test()
   runs on MTCType system SystemType {
   ...
   cp.send("has_stock") to user1;
   cp.receive("ack") from user1;

   cp.send("has_stock") to user2;
   cp.receive("ack") from user2;

   cp.send("has_stock") to user3;
   cp.receive("ack") from user3;

   // after three purchase requests,
   // the item is now out of stock

   cp.send("out_of_stock") to user4;
   cp.receive("ack") from user4;

   cp.send("out_of_stock") to user5;
   cp.receive("ack") from user5;

   ...
}
```

Figure 7. Test coordination by MTC

 ISBN: 978-1-61208-776-4

In TTCN-3, the *receive* statement is blocking. Thus, the rest of the behavior of the PTC will not execute while the coordination message has not been received.

Note the returned *ack* message. The *ack* is used to prevent racing. In other words, a new individual test cannot occur before the previous test has fully completed, otherwise more requests are being sent to the server which may change its state before a response is sent back to a user resulting in failure. We have observed that removing the ack effectively produces race conditions. We leave this verification as an exercise to the reader.

2) Coordination using PTC Threads operations

PTCs are in fact translated by the TTCN-3 compiler that produces an executable in a general purpose language (GPL) such as Java or C++ and many others using threads. Thus, one typical Thread operation that is available in TTCN-3 is to check if the thread has terminated. This is represented in TTCN-3 with the keyword *done*. Here, as shown in Figure 8, each PTC is started using a parameter representing the function behavior that carries the predicted state of the SUT.

There are in fact two ways to use this feature: the first one consists in placing the done statement immediately after the corresponding start statement. This would result in transforming a concurrent system into a sequential execution system with effects similar to the coordination messages solution shown in the previous section.

```
testcase thread_operations_test()
   runs on MTCType system SystemType {
   …
   user1.start(purchasingBehavior
                       ("has_stock"));
   user2.start(purchasingBehavior
                       ("has_stock"));
   user3.start(purchasingBehavior
                       ("has_stock"));

   user1.done;
   user2.done;
   user3.done;

   user4.start(purchasingBehavior
                     ("out_of_stock"));
   user5.start(purchasingBehavior
                     ("out_of_stock"));
   user6.start(purchasingBehavior
                     ("out_of_stock"));

   user4.done;
   user5.done;
   … }
```

Figure 8. MTC behavior using PTC threads operations

In this second approach, we have chosen to place all the done statements after all the start statements for the first three PTCs to simulate the database reaching its maximum inventory. This has the advantage to at least conserve some of the concurrent behavior of the system and thus avoiding a full sequential test execution of PTCs.

3) Introducing semaphores to TTCN-3

In a way the second approach is less sequential than the first one but still somewhat sequential. Thus, we have explored a third solution that would eliminate some aspects of the sequential aspect of this test behavior. The method consists in using shared variables and semaphores among PTCs. The shared variable keeps track of the inventory on hand and enables a PTC to determine the state of the SUT on its own. However, TTCN-3 does not have the concept of shared variables, neither semaphores and thus we recommend modifying the standard. In our case, we need to declare the inventory variable as shared. TTCN-3 test suites are always translated in a GPL that is then compiled and executed. Since this feature is not available in TTCN-3 we have used an implementation in Java that would be typically comparable to the one generated by the TTCN-3 compiler but somewhat simplified to make it easier to understand.

```
public static void main(String args[])
      throws InterruptedException {

   PTCtype ptc1 = new PTCtype("ptc1");
   PTCtype ptc2 = new PTCtype("ptc2");
   PTCtype ptc3 = new PTCtype("ptc3");
   PTCtype ptc4 = new PTCtype("ptc4");
   PTCtype ptc5 = new PTCtype("ptc5");
   PTCtype ptc6 = new PTCtype("ptc6");

   ptc1.start();
   ptc2.start();
   ptc3.start();
   ptc4.start();
   ptc5.start();
   ptc6.start();

   ptc1.join();
   ptc2.join();
   ptc3.join();
   ptc4.join();
   ptc5.join();
   ptc6.join();

  }
```

Figure 9. MTC behavior using semaphores

Note that the java main method of Figure 9 corresponds to the TTCN-3 MTC test case behavior. The basic difference with the TTCN-3 version shown in

 ISBN: 978-1-61208-776-4

Figure 8 is the presence of the join method that needs to be added to the TTCN-3 standard and the absence of state indication sent to the PTCs. The *join* statement does not exist in TTCN-3 and is part of our recommendation in modifying the standard. Now, we need to show the different modification required in the definition of the PTCs behavior as shown in Figure 2 to implement the semaphores. The PTC type needs first to declare a semaphore as does the Java version. The new TTCN-3 Semaphore data type would merely be translated to the corresponding Semaphore class in Java.

```
class PTCType extends Thread {
      Semaphore sem;
      String threadName;
      String state = "";
```

Then the Semaphore instance needs to have an *acquire* statement as in Java:

```
      sem.acquire();
```

The shared variable *inventory* is then used to compute the predicted state that can be used in the TTCN-3 *alt receive* statement:

```
      if(inventory > 0) {
         inventory--;
         state = "has_stock";
      }
      else
         state = "out_of_stock";

      // place the alt statements as shown on Figure 2
here.
```

And finally add a semaphore *release* statement at the end of the PTC behavior as follows:

```
      sem.release();
```

4) Evaluation

We have observed that the semaphore version of this problem produces a sequence of execution very similar to the first approach using coordination messages. The only difference being that the sequence of the executed PTCs is not entirely in the order of the start of each PTC, i.e. from 1 to 6. Instead the semaphore version produces various sequences of PTC execution but in all remain sequences thus preventing discovering concurrency problems. Thus, we think that the second approach that consists in running PTCs in batches of states is possibly a better approach. However, the second method may run into problems when complex templates are used for depicting for example shopping baskets where the various items may have different limits. In any case, this method is much better than unit testing.

III. TTCN-3 AS A MODELLING LANGUAGE

Normally, testing activities can take place only once the SUT has been fully developed and is runnable. However, planning and developing automated test cases can be done in parallel to the SUT development phase. More importantly, the missing SUT can be emulated using TTCN-3. This enables us to find any flaws in the automated test suites before we apply them to the SUT and thus reduce time to market.

In our case study, this means finding a way to portray a behavior that replies with “invoice” when there is inventory on hand and replies “out of stock” when inventory has reached zero. At the abstract level, there is no need to implement a full system, in our case probably a web application and a related database. The implementation of such an abstract system is as follows:

```
function SUTbehavior() runs on SUTType
{
   var integer inventory := 3;
   var PTCType ptc := null;
   var MTCType mtc_ := null;

   alt {
      [] p.receive("purchase") ->
                          sender ptc {
         if(inventory > 0) {
            p.send("invoice") to ptc;
            inventory := inventory -1;
         }
         else {
            p.send("out_of_stock") to
                                  ptc;
         };
         repeat
      }
      [] ap.receive("stop")
                  -> sender mtc_
         setverdict(pass)
      }
   }
}
```

Figure 10. SUT behavior

We use a simple variable to portray the inventory that we set at 3 units. Every time a request to purchase an item comes in, we decrease the inventory. A simple if-then-else statement provides the correct response of *invoice* or *out-of-stock* state. At the abstract level, this is all we need.

Also, the test suite is developed in two different levels of abstraction. First, we use simplified messages like here simple strings with values. Once we simulate the abstract system and we are happy with the results, in a second step we merely redefine the abstract data types and its corresponding templates (test oracles for received messages and data content for sent messages) as follows:

 ISBN: 978-1-61208-776-4

1st step: Data types and templates declarations:

```
type charstring RequestType;
type charstring ResponseType;

template RequestType myRequest_t :=
                        "purchase";

template ResponseType
           myInvoiceResponse_t
                        := "invoice";
template ResponseType
           myOutOfStockResponse_t:=
                    "out_of_stock";
```

Figure 11. simplified data types and templates

2nd step: Real data types and templates:

```
type record RequestType {
    charstring bookName,
    charstring ISBN
}

type record ResponseType {
    charstring bookName,
    charstring ISBN,
    charstring status,
    charstring action
}

template RequestType myRequest_t := {
    bookName := “ttcn-3 in a
nutshell”,
    ISBN := “978-2-345-678”
}

Template ResponseType myResponse_t :=
{
      bookName := “war and peace”,
      ISBN := “978-2-345-678”,
      Status := “on hand”,
      Action := “invoice”
}
```

Figure 12. Fully realistic data types and templates

Note that both datatypes and templates are defined using the same identifiers. Only their content is different.

IV. CONCLUSION

Despite its long history, testing concurrent systems remains complex and does not always provide accurate results. In this paper we have shown that using formal methods for testing such as TTCN-3 helps to locate problems accurately because of the wide choice of results visualization features that the various commercial and open source editing, and execution tools provide. We also recommended enhancing the TTCN-3 standard by providing shared variables and semaphore features for the MTC and the PTCs. We also have shown a way to partly avoid sequencing PTC test by using batches of concurrent tests by using the current features of TTCN-3.

ACKNOWLEDGMENT

The authors would like to thank NSERC for funding this research.

REFERENCES

[1] ETSI ES 201 873-1, The Testing and Test Control Notation version 3 Part 1: TTCN-3 Core Language, May 2017. Accessed March 2018 at http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.09.01_60/es_20187301v040901p.pdf

[2] TTworkbench,Spirent, https://www.spirent.com/Products/TTworkbench

[3] Titan, https://projects.eclipse.org/proposals/titan

[4] E. Boros and T. Unluyurt, Sequential Testing of Series-Parallel Systems of Small Depth in ISBN 978-1-4613-7062-8

[5] A. Bertolino, Software Testing Research: Achievements, Challenges, Dreams in proceedings of FOSE ’07 pp 85-103

[6] T. Hanawa, T. Banzai, H. Koyzumi, R. Kanbayashi, T. Imada and M. Sato, Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems in 2010 Third International Conference on Software Testing, Verification and Validation Wokshops procedings

[7] A. M. Alghamdi and F. Eassa, Software Testing Techniques for Parallel Systems: A Survey in IJCSNS International Journal of Computer Science and Network Security, vol 19. No 4, April 2019, pp 176-184

[8] L. Parobek, 7 Reasons to Move to Parallel Testing in white paper on https://devops.com/7-key-reasons-make-move-sequential-parallel-testing/

[9] B. Rao G. , K. Timmaraju, and T. Weigert, Network Element Testing Using TTCN-3: Benefits and Comparison in SDL 2005, LNCS 3530, pp. 265–280, 2005

[10] G. Din, S. Tolea, and I. Schieferdecker, Distributed Load Test with TTCN-3, in Testcom 2006 proceedings, pp 177-196

[11] B. Stepien, K, Mallur, L. Peyton, Testing Business Processes Using TTCN-3, in SDL Forum 2015 proceedings, Lecture Notes in Computer Science, vol 9369. Springer, Cham.

 ISBN: 978-1-61208-776-4