Semaphore Implementations for Testing Concurrent Systems using TTCN-3

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

Jacob Wieland, Dirk Tepelmann, Dirk Borowski
Spirent Communications
Berlin, Germany
Email: {Jacob.wieland | dirk.tepelmann | dirk.borowski}
@spirent.com

Abstract—**Testing concurrent systems is a complex task. 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. An easy solution to this problem is to use semaphores that avoid race conditions. Since semaphores do not exist in TTCN-3, we have explored solutions using the TTCN-3 concept of external functions. This allows us to define behavior in the run-time language used by the TTCN-3 compiler, in our case Java, without having to modify the TTCN-3 standard. However, Java semaphores can block other parallel processes which may actually defeat parallelism. Thus, we have explored other solutions based on resource blocking rather than process blocking. This allows two or more concurrent processes to perform operations on different resources and thus achieve more sophisticated concurrency testing.**

Keywords- software testing-concurrent systems; TTCN-3; test oracles; race conditions; semaphores.

I. Introduction

This paper is a significant extension and update of our previously published SOFTENG 2020 paper that presented preliminary results for advanced techniques for testing concurrent systems [1].

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 [2]. 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 [3][4][5][6][7]. Obviously there are different kinds of parallel testing. In the previous references, 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 (SUT) as the test purpose. There are two main categories of concurrent testing:

- Response time testing when a large number of requests are sent to a server as shown in Figure 1. Techniques for addressing this category of concurrent testing using TTCN-3 are presented in [8].
- Testing the 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.

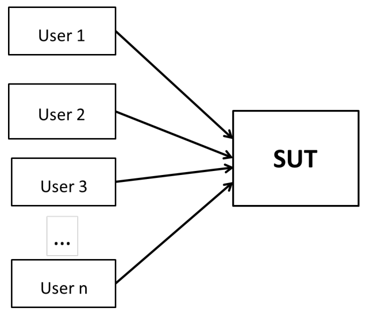


Figure 1. Parallel system configuration

In this paper, a case study, using the formal test specification language TTCN-3[9], illustrates the challenges for test coordination for this category of testing, especially with respect to 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 test specification language such as TTCN-3 and its related available execution tools 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 by introducing semaphores as a built-in language feature to make our testing concurrency problem statement usable.

There are two kinds of race conditions:

- Shared variables racing
- Behavior racing

A. *Shared variables racing*

The only way to predict a state of a SUT when multiple users are present is to use a shared variable to record for example the state of the inventory of a given product each user is trying to order. Each time a user orders a product, the inventory is decreased by one unit. However, in concurrency, a user trying to make a decision on the state of the SUT may have the shared variable overwritten by another user, thus producing a wrong result. Here the solution would be to lock the shared variable so that no other user can overwrite it.

B. *Behavior racing*

We have determined that shared variable locking is actually not sufficient to solve the race condition problem. We determined through experimentation that some sequence of events must also be locked. For instance, once the expected state of the SUT is determined using the shared variable, we must prevent another user to place a request for the same product to the SUT. The SUT also must decrease its inventory of the product. Thus if another user places an order while executing the actual purchase of a product, this will decrease the inventory and the calculated inventory by the first user will be out of synch with the real inventory of the SUT once the second user has interfered.

II. A Case Study

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

A. *Defining the problem*

Although we have studied extensively testing of concurrency problems in industrial applications [10][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: an invoice and shipping details if it has stock; or an out of stock notification. 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 individually. 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 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 predicted.

The interesting aspect of this simple example is that we have tried various approaches to coordination and some resulted in race condition problems, thus disturbing the test process altogether. Table I shows the values of test oracles depending on 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.

Most testers use unit testing that is simple but misses some aspects of the test. Unit testing would consist of putting the SUT in the appropriate state and check the individual responses to a request.

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 test case 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. In Figure 2, the state variable must be provided. This happens either as setting its value while starting the PTC or computing it as the PTCs place their orders.

```
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.

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

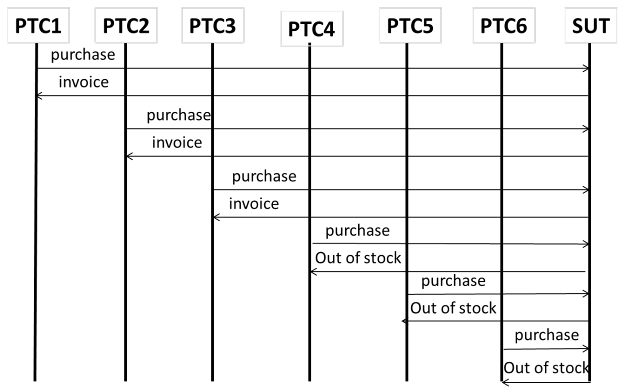


Figure 3. Ideal testing responses

This enables the tester to locate rapidly the point of failure and investigate the problem rapidly.

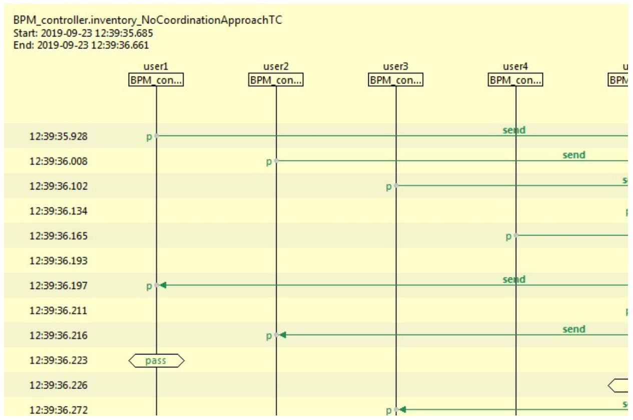
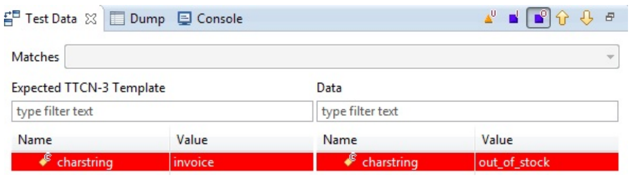


Figure 4. Uncoordinated execution results

Figure 5 shows the TTCN-3 tools data inspection feature that provides detailed message and test oracle contents that enable the tester to understand the reasons for failure especially when complex data types with many fields are used.



Expected TTCN-3 Template

Name	Value
charstring	invoice

Data

Name	Value
charstring	out_of_stock

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.

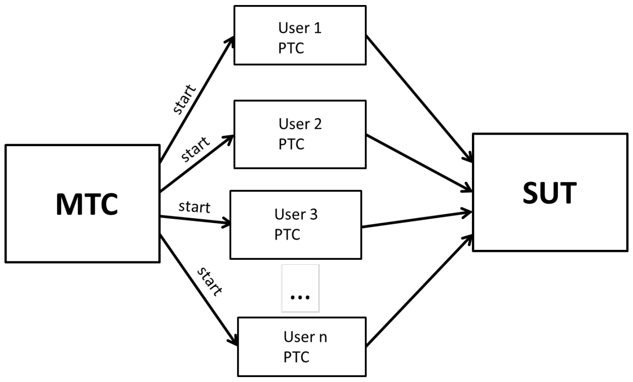


Figure 6. Test coordination with MTC

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 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

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 behavior 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 for 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 the zero inventory point. 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 Java 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, nor semaphores. We have explored how we could implement semaphores in the Spirent TTworkbench tool [12] using the TTCN-3 concept of external functions to link the TTCN-3 abstract behavior description to functions that are written in a GPL (e.g., Java in our case). This avoids changing the ETSI standard which would require a lengthy approval process.

III. Implementing Shared Variables and Semaphores in TTCN-3

In Java, shared variables are implemented using independent classes that contain the shared variable as an attribute. In TTCN-3, there is a simple way to reproduce this model using TTCN-3 user defined data types that allow the creation of PTCs that are actually translated into Java classes by the compiler.

For semaphores, we will use two different types of approaches:

- A semaphore data type related to an external function written in Java that creates an instance of the Java Semaphore class.
- Our own definition of semaphores directly in TTCN-3.

The second approach is for the purpose of using a more flexible kind of semaphore as opposed to the Java semaphore that blocks any process that did not acquire the semaphore. The second approach allows blocking individual resources rather than processes (here PTCs).

A. *implementing shared variables*

This is achieved by creating a datatype to keep track of the inventory. In this case we are trying to keep track of the inventory of two different products, product_A and product_B.

```
type component InventoryCompType
{
   var integer inventory_A := 3;
   var integer inventory_B := 4;
}
```

This data type is then used to create an instance of a PTC that will receive requests from users, compute the state of the inventory and reply with that state. Therefore we need to add a communication port that will be used to receive or send messages with the users:

```
  port CoordPortType ip;
```

The basic inventory component is found on Figure 9. It is composed of two groups of events:

- Receive a request from a user
- Compute the actual state of the inventory
- Return that state to the user

```
function InventoryBehavior()
          runs on InventoryCompType {

var InventoryCompType user;

alt {
  [] ip.receive("purchase_A")
                -> sender user {

     if (inventory_A > 0) {
      ip.send("has_stock") to user;

      inventory_A := inventory_A - 1;
     }
     else {
      ip.send("out_of_stock") to user;
     }
     repeat;
  }
  [] ip.receive("purchase_B")
                -> sender user {

     if (inventory_B > 0) {
      ip.send("has_stock") to user;
      inventory_B := inventory_B - 1;
     }
     else {
      ip.send("out_of_stock") to user;
     }
     repeat;
  }
  [] api.receive("stop") {
     setverdict(pass);
  }
 }
```

Figure 9. Inventory computation behavior

B. Test configuration

There are four basic types of parallel test components as shown in Figure 10:

- The master test component (MTC)
- The inventory component
- The users components
- The System Under Test component (SUT)

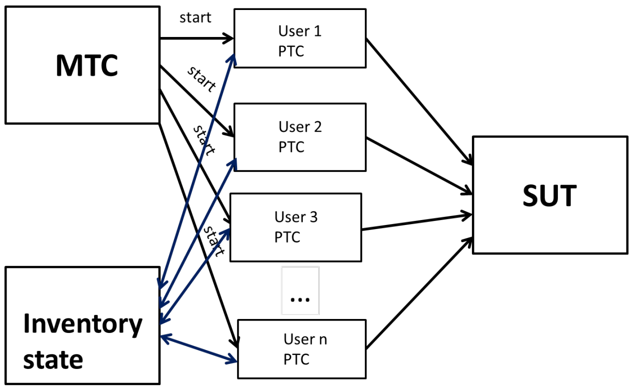


Figure 10. Shared variable handled in a separate PTC

The SUT component defined as follows:

```
type component SUTType {
      port MyPortType up;
      port MyAdminPortType aps;
      port MyAdminPortType api;
}
```

The port *up* will carry all the communication messages between the SUT and the individual user PTCs. The admin port *aps* is used at the very end of the test in order to terminate the SUT that otherwise is in a constant loop trying to receive messages from the users. The port api is used to terminate the inventory PTC that is in a similar continuous loop.

The user's component is defined as follows:

```
type component PTCType {
      port MyPortType sp;
      port InventoryPortType ip;
}
```

The port *sp* is used to communicate with the SUT while he port *ip* is used to communicate with the inventory PTC to inquire about the state of the inventory.

Finally, we define the master test component (MTC) as follows:

```
type component MTCType {
      port MyAdminPortType aps;
      port MyAdminPortType api;
}
```

The port *aps* is used to communicate with the SUT while the port *api* is used to communicate with the inventory PTC.

C. Testcase configuration under Java semaphores using TTCN-3 external functions

A TTCN-3 test case in our particular PTC configuration is shown in the following TTCN-3 code. As shown in Figure 11, it consists of creating an instance of a Semaphore object in the MTC that is passed on to each created user PTC and connections between the MTC and each instance of user PTCs. This enables a user to acquire or release that centralized Semaphore object.

```
Testcase
   inventory_semaphore_approach()
   runs on MTCType system SystemType {

   var Users user;
   var SemaphoreType semaphore;

   semaphore := Semaphore.new();
```

This instance of Semaphore is then passed on to each user PTC when starting their behavior function.

```
   for (var integer i:=0; i <nb_users;
                            i:=i + 1) {
      user[i].start(
          purchasingBehaviorSemaphore(
                            semaphore))
   }
```

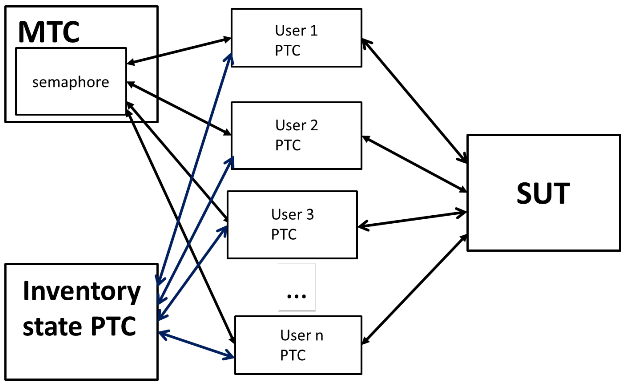


Figure 11. Java semaphore integration

In creating a SUT PTC and each user PTC and starting them using a specific behavior function, we also create an inventory PTC that keeps track of the inventory level for each product and that is connected to each PTC that query it.

TTCN-3 is based on strong typing. Thus, while the TTCN-3 source code is converted to Java, semaphores in TTCN-3 need to be typed. The translator then maps the two versions automatically. The definition of TTCN-3 semaphore is based entirely on external functions definitions as follows:

First the abstract definition of semaphores:

```
module Semaphore {
  type integer SemaphoreType;

  external function new()
        return SemaphoreType;

  external function acquire
            (SemaphoreType semaphore);

  external function release
            (SemaphoreType semaphore);
}
```

We have explored the approach of using the existing external functions definition feature of TTCN-3 to link the above abstract semaphore to the underlying Java semaphores as shown in Figure 12.

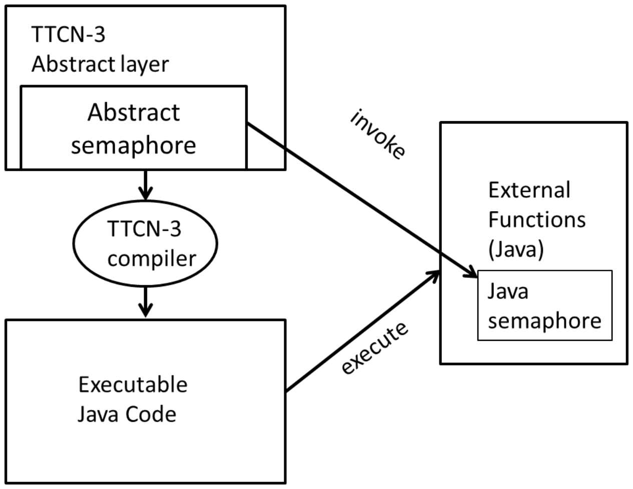


Figure 12. TTCN-3 external functions

The external function Java code is as follows:

```
package com.spirent.externalfunctions.
                            semaphore;
import java.util.HashMap;
import java.util.concurrent.Semaphore;
import org.etsi.ttcn.tri.TriStatus;
import com.testingtech.ttcn.annotation
                  .ExternalFunction;
import com.testingtech.ttcn.tri.
    AnnotationsExternalFunctionPlugin;

  @ExternalFunction.Definitions(
                 ExtSemaphore.class)
  public class ExtSemaphore extends
   AnnotationsExternalFunctionPlugin {

   private HashMap<Integer, Semaphore>
         semaphores = new HashMap<>();

   @ExternalFunction(name = "new",
              module = "Semaphore")
   public int newSemaphore() {
     int result = semaphores.size();
     semaphores.put(result,
                  new Semaphore(1));
     return result;
   }

   // external function acquire(
             SemaphoreType semaphore);
   @ExternalFunction(name = "acquire",
              module = "Semaphore")
   public void acquire(int semaphore)
   {
      try {

        semaphores.get(semaphore).
                            acquire();
      } catch (InterruptedException e)
        {throw new
                RuntimeException(e);
        }
    }

    // external function release
            (SemaphoreType semaphore);
   @ExternalFunction(name =
      "release", module = "Semaphore")
   public void release(int semaphore)
   {
      semaphores.get(semaphore).
                            release();
   }

   @Override
   public TriStatus tearDown() {
      semaphores.clear();
      return super.tearDown();
   }
}
```

While executing the test case based on external Java Semaphores, we have noticed that the results look like another type of sequence comparable to our first approach shown on Figure 3, with the only difference a random order of execution of user PTCs behavior and the calculation of the inventory state rather than the hardcoded state used in this first approach.

The user behavior is merely framed by the invocation of the semaphore acquire and release statements as shown in Figure 13.

```
function purchasingBehaviorSemaphore(
              SemaphoreType semaphore)
                     runs on PTCType {
  var charstring state;

  timer t := rnd();
  t.start;
  t.timeout;

  Semaphore.acquire(semaphore);

  //  communication with inventory PTC

  ip.send("purchase_A");
  ip.receive(charstring:?) ->
                        value state;

  // communication with SUT

  sp.send(request_product_A_t);

  alt {
    [state == "has_stock"]
     sp.receive(myInvoiceResponse_t) {
     setverdict(pass);
    }
    [state == "out_of_stock"]
     sp.receive(myInvoiceResponse_t) {
     setverdict(fail);
   }
   [state == "out_of_stock"]
  sp.receive(myOutOfStockResponse_t) {
     setverdict(pass);
  }
  [state == "has_stock"]
  sp.receive(myOutOfStockResponse_t) {
     setverdict(fail);
  }
 };

 Semaphore.release(semaphore);
}
```

Figure 13. User behavior with Java semaphore

D. Semaphores defined in TTCN-3

Effectively, the Java semaphore blocks completely the execution of user PTCs that didn't acquire the semaphore as of yet. Here we have determined that this does not provide a full concurrency behavior. Thus, instead, we have focused on a resource-oriented semaphore that would block the inventory rather than the user processes. This would allow a user to purchase product A while another user would purchase product B. There are no race condition in such a case.

In this approach, we need a mechanism in the inventory PTC to block its access by any PTC that did not successfully acquire the semaphore.

The test case is similar to the one used for the Java semaphore example with one major difference, there is no instance of a Java semaphore. Our own semaphore is actually implemented in the Inventory behavior where the inventory shared variables are protected.

1) User behavior

The user no longer uses the Java semaphore but instead invokes our own semaphore using TTCN-3 procedure invocations (call acquire) as shown on Figure 14. Since TTCN-3 procedures are non-blocking, the semaphore call will wait until the resource (inventory) is released again. This is indicated by the getreply statement that is triggered only if the call to acquire is accepted. The rest of the user behavior is identical to the one used for Java semaphores.

```
Function
     purchasingBehaviorInventoryComp()
                 runs on InventoryPTC {
  var charstring state;

  semaphore.call(acquire:{}, 100.0) {
      [] semaphore.getreply {}
      [] semaphore.catch(timeout) {
            setverdict(inconc, "could
              not acquire semaphore");
            return;
      }
  }

  inventory.send("purchase_A");
  inventory.receive(charstring:?) ->
                           value state;
  p.send(request_product_A_t);
  alt {
   [state == "has_stock"]
     p.receive(myInvoiceResponse_t) {
      setverdict(pass);
     }
   [state == "out_of_stock"]
     p.receive(myInvoiceResponse_t) {
      setverdict(fail);
     }
   [state == "out_of_stock"]
     p.receive(myOutOfStockResponse_t)
                                    {
      setverdict(pass);
     }
   [state == "has_stock"]
      p.receive
           (myOutOfStockResponse_t) {
        setverdict(fail);
      }
   }
 }

 semaphore.call(release:{});
}
```

Figure 14. user behavior with our definition of semaphores

2) Use of semaphores in the inventory PTC

The procedures acquire_A and release_A and their corresponding names for product B are residing in the inventory behavior. Here we are using the variables inventory_A_blocked and its counterpart for B to control which part of the inventory calculation code can be reached.

```
function InventoryBehavior()
          runs on SemaphoreInventory {
  var integer blockedBy_A := -1;
  var integer blockedBy_B := -1;
  var InventoryPTC user;
  var boolean inventory_A_blocked :=
                              false;
  var boolean inventory_B_blocked :=
                              false;

  alt {
   [inventory_A_blocked == false] any
     from sem_p.getcall(acquire_A:{})
         -> @index value blockedBy_A {

       inventory_A_blocked := true;

       sem_p[blockedBy_A].reply(
                 acquire_A:{});
       repeat;
   }
   [inventory_A_blocked == true]
       sem_p[blockedBy_A].getcall(
                    release_A:{}) {

     blockedBy_A := -1;

     inventory_A_blocked := false;
     repeat;
   }
   [inventory_B_blocked == false] any
     from sem_p.getcall(acquire_B:{})
        -> @index value blockedBy_B {

     inventory_B_blocked := true;
     sem_p[blockedBy_B].reply(
                     acquire_B:{});
                       repeat;
   }
   [inventory_B_blocked == true]
         sem_p[blockedBy_B].getcall(
                     release_B:{}) {
     blockedBy_B := -1;
     inventory_B_blocked := false;
     repeat;
   }

   [inventory_A_blocked == true]
    inventoryPort.receive(
     "purchase_A") -> sender user {
     if (inventory_A > 0) {
       inventoryPort.send(
             "has_stock") to user;

       inventory_A := inventory_A -
1;
     }
     else {
        inventoryPort.send(
           "out_of_stock") to user;

          repeat;
     }
   [inventory_B_blocked]
     inventoryPort.receive(
       "purchase_B") -> sender user {

     if (inventory_B > 0) {
      inventoryPort.send("has_stock")
                             to user;

      inventory_B := inventory_B - 1;
     }
     else {

      inventoryPort.send(
                    "out_of_stock")
                            to user;
      }
      repeat;
    }
    []  api.receive("stop") {
        setverdict(pass);
    }

  }
}
```

Figure 15. Inventory behavior with TTCN-3 semaphores

The execution of a single component's behavior produces the sequence of events shown on Figure 16. We can clearly observe that when the inventory PTC replies with the *has_stock* message, the actual purchase message sent to the SUT results in an invoice coming back from the SUT.

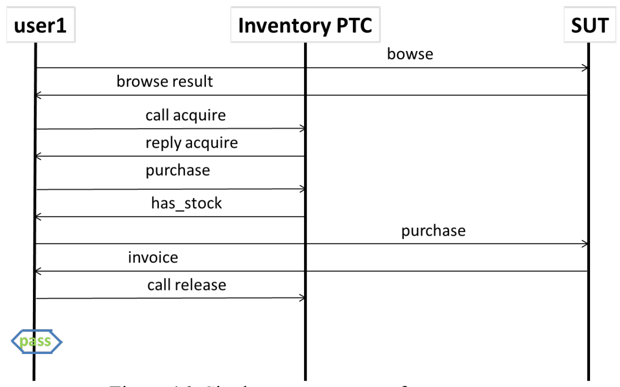


Figure 16: Single user sequence of events

E. *Mutually Exclusive Behavior Blocks*

As can be seen easily, this way of modeling semaphores is very complicated and the code obfuscates the intent, reducing the manageability.

We therefore propose to add a new feature to TTCN-3 that allows controlled, shared access to component variables from the scope of other component behaviors.

This would be syntactically modelled by an on-statement of the following form:

on *ComponentRef* [**@readonly**] *StatementBlock*

The *StatementBlock* would be allowed to access component variables of the component referenced via *ComponentRef* via dotted notation.

The on-statement uses an implicit read lock (in case of @readonly) or read-write lock behavior otherwise of a re-entrant lock associated with the referenced component. It is assumed that there is one such re-entrant execution lock associated with every created component. If the read-write lock is acquired by any component, no other component can acquire the lock of that component. If the read-lock of a component is acquired by any component, the read-write lock of the component can not be acquired.

Components trying to acquire a lock that can currently not be acquired block their execution until the lock becomes available again or their behavior is terminated.

The read-write lock is acquired whenever the component starts executing behavior or when a non-read only on-statement for that component is entered. It is released when the component terminates its current behavior or starts waiting in an alt statement or when the outermost on-statement referencing the component in the behavior of another component is left.

The read-lock of a component is acquired whenever a read-only on-statement for that component is entered. Thus, it is possible that several components read the component variables of one component in parallel or that a single component has read-write access to the referenced components variables at any one time.

The advantage of this more declarative approach is that the code is more readable and that it is possible to statically analyse the types and names of the used variables, whether they are only read in read only on-statements.

Using this construct, the inventory component becomes a simple shared variable container without any executed behavior and is passed as a reference to each PTC for referencing when starting the following behavior.

```
function purchasingBehaviorInventory
(InventoryComp inventory)
runs on PTCType {
  on inventory {
  // acquired read-write lock of
  // inventory
    var charstring state;
    if (inventory.inventory > 0) {
      state := "has_stock";
      inventory.inventory :=
        inventory.inventory - 1;
    } else {
      state := "out_of_stock";
    }
    p.send(myRequest_t);
    alt {
    [state == "has_stock"]
      p.receive(myInvoiceResponse_t) {
        setverdict(pass);
      }
    [state == "out_of_stock"]
```

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

     p.receive(myOutOfStockResponse_t) {
          setverdict(fail);
     }
      }
   } // release read-write lock of
     // inventory
}
```

F. *Mixing blocked and free behavior*

Testing web applications has been studied intensively for several decades. An early attempt can be found in [14]. However, none of them mention race conditions.

So far, we have shown the simple behavior of users that create race conditions. This is achieved by blocking shared variables and behavior sequences. However, events in an e-commerce application are not always subject to race condition. The race condition occurs only when a user is trying to make a purchase and thus the state of the inventory must be determined. On the other hand, browse events are not subject to race conditions at all. They can occur any time without affecting inventories states. Thus, we have created an example where browse events precede the purchasing event and we have introduced sequences of different purchase events.

With several users and the alternate blocked and unblocked behaviors of users we may end up with various sequences of events interleaving between the users. The sequence in Figure 17 shows that user2 places its order of product A before user1 places its order for product B. The actual specification of a user's behavior may suggest that a given user may place orders for product A and product b in strict sequence without interruption by another user but this figure clearly shows that this is not the case. Here the unblocked browse event allowed such an interleaved sequence. However, it is to be noted that block portions of behavior remain together and thus cannot be interrupted by another user.

IV. 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.

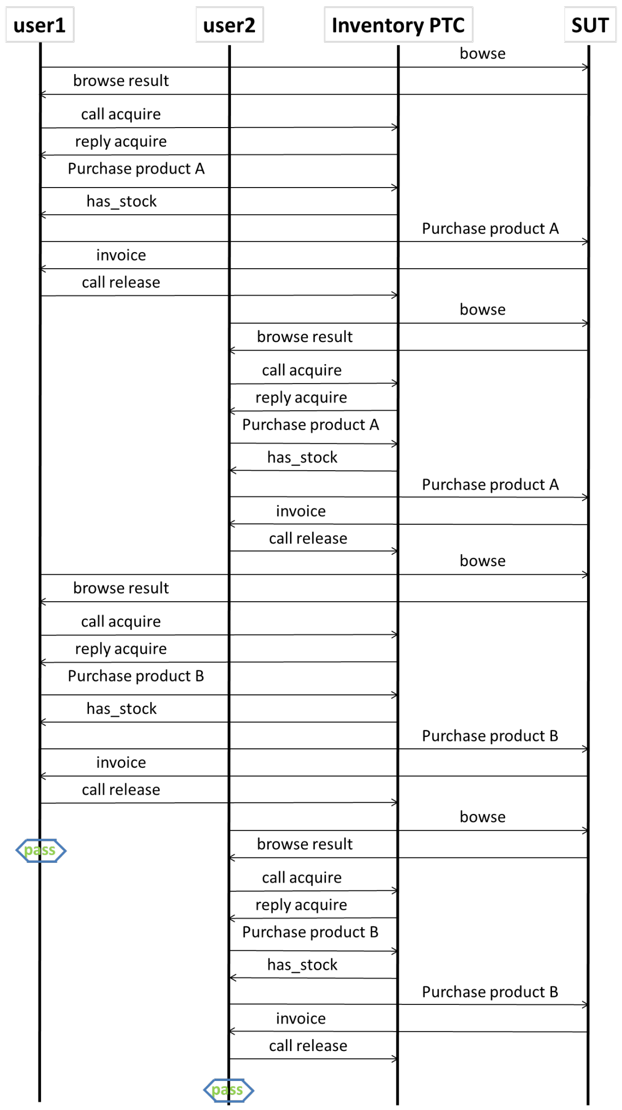


Figure 17. Multiple users: possible sequence of events

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 18. 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 simple strings. 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:

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 19. 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 20. Fully realistic data types and templates

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

V. 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 have experimented with the TTCN-3 external functions concept in order to implement shared variables and Java semaphore features for the MTC and the PTCs. We have discovered that the traditional Java semaphores approach prevents true concurrency testing and we have developed a system that allows blocking resources rather than processes to avoid racing conditions which provides a considerably more flexible concurrency testing.

Acknowledgment

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

References

[1] B. Stepien and L, Peyton, Test Coordination and Dynamic Test Oracles for Testing Concurrent Systems, in Proceedings of SOFTENG 2020

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

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

[4] 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

[5] 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

[6] L. Parobek, 7 Reasons to Move to Parallel Testing in white paper on https://devops.com/7-key-reasons-make-move-sequential-parallel-testing/, last accessed December 14th, 2020

[7] 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

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

[9] 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

[10] 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. Pp. 252-267

[11] B. Stepien, L. Peyton, M. Shang and T. Vassiliou-Gioles, An Integrated TTCN-3 test framework architecture for interconnected object-based applications in IJEB vol. 11, No. 1, 2014

[12] TTworkbench,Spirent, https://www.spirent.com/Products/TTworkbench, last accessed December 14th, 2020

[13] Titan, https://projects.eclipse.org/proposals/titan, last accessed December 14th, 2020

[14] F. Ricca and P. Tonella, Testing Processes of Web Applications in Annals of Software Engineering 14, pp. 93-114 (2002)