TTCN-3 Snapshot implementation

Julien Deltour - Emmanuel Gaudin
PragmaDev
julien.deltour @ pragmadev.com - emmanuel.gaudin @ pragmadev.com

Abstract. Implementation of TTCN-3 in RTDS tool raised some interrogations. The concept of snapshot in the TTCN standard is not present in traditional RTOS. This paper will first make a short presentation of TTCN-3. It will then explain the *alternative* rationale and the concept of snapshot. A reminder of event-driven concepts in major RTOS will be done, to finish with a presentation of several solutions to implement the snapshot.

Introduction

TTCN-3 is a testing language standardized by the ITU-T. A TTCN-3 test suite is composed of several modules. Modules are composed of an optional control part and some test cases.

Test cases run on components. A component defines the interface of the running element (set of ports). These components can communicate with each other through ports. Each test case is organized with a Master Test Component (MTC), a set of Parallel Test Component (PTC) and a System Interface with the System Under Test (SUT).

Test case behavior is a suite of operations, including communication operations such as timer timeout, message receive, test on component state, and so on. These operations can be sequentially expressed, and in this case, each one is tested one after the other.

The *Alternative* rationale

It is also possible, that at some point in the test execution, several events might occur. In this case, the alternative statement can be used. The *alt* statement allows to combine several communication operations. Each operation will be tested, and the first one which can proceed will determine the continuation of the test execution.

```
alt {
        []p1.receive(m1){
                …
                };
        []p2.receive(m2){
                …
                };
        []timer1.timeout{
                …
                };
        }
```

Each operation of the *alt* statement have to be tested with exactly the same information. That is why it is necessary to save every relevant data before alternatives are evaluated: the snapshot.

Execution of an alternative behavior:

When entering an *alt* statement, a snapshot is taken. A snapshot is a partial view of the current component that includes every information necessary to evaluate boolean conditions and alternatives: timeout events, stopped and killed components, pending messages, calls, replies and exceptions. Each alternative is evaluated in the order of their appearance.

Boolean guard: Before each alternative, it is possible to evaluate a boolean expression based on the snapshot. There is one special guard [else]. The Else-branch will always be chosen when none of other branch has been selected.

```
alt {
        [x>2]p1.receive(m1){
                …
                };
        [x<2p2.receive(m2){
                …
                };
        [else]{
                …
                };
        }
```

If the boolean guard is fulfilled, the alternative will be evaluated.

When an alternative has been selected, execution continues with the statements list of chosen alternative and then leaves the alt statement. It is possible to re-evaluate alt statement again by using the **repeat** statement. In that case if none of the alternative branches has been selected, a new snapshot is taken, and all the guards and alternatives are re-evaluated.

The test shall stop and raise an error if the component is totally blocked, meaning no relevant component and no relevant timer are running, and all relevant ports contain at least one message, call, reply or exception.

Basic rules of event-driven RTOS

Event-driven or event-based RTOS are operating system in which the flow of execution is determined by events reception. Each process or thread will usually process an action and wait for an incoming event. This event can be a message reception or a timer going off for example. Behavior of an event driven process is express sequentially, and only one port of event can be read at a time. Therefore, in opposition to TTCN, alternative statements are not naturally present in event-driven RTOS, neither is the concept of snapshot. Last but not least, most RTOS give only access to the first message in each reception queue.

How to implement the alt statement and the snapshot mechanism

As seen before, alternative statements are not supported by most RTOS. For our TTCN to C code generator, we had to find a solution to properly implement this concept. Many solutions have been considered, each one with its pros and cons.

Solution 1: Test every branches

The first solution consist in simply test every branches of the alt statement the one after the other. For example, in the following alt statement:

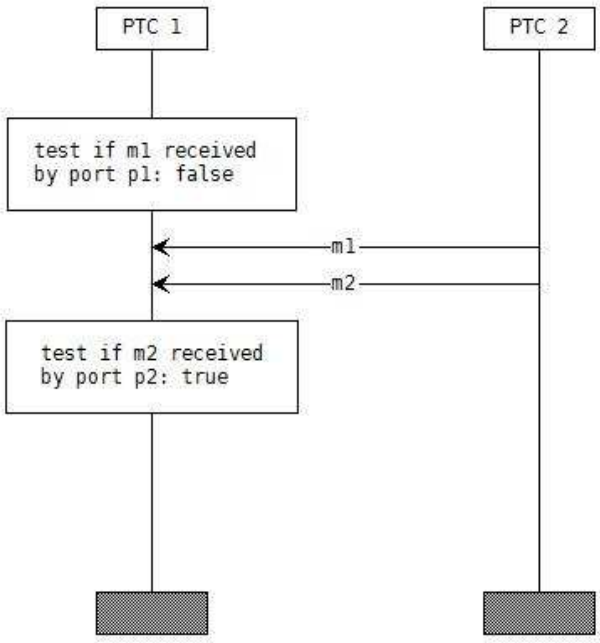
```
alt {
        []p1.receive(m1){
                ...
                };
        []p1.receive(m2){
                ...
                };
        []p2.receive(m2){
                ...
                };
        }
```

We will first check if a message is present in port1 receive queue, and then match it with m1. Depending on the result of this test, the test execution will continue or an other branch of the alt statement will be test.

Problems: Many problems appear with this solution.

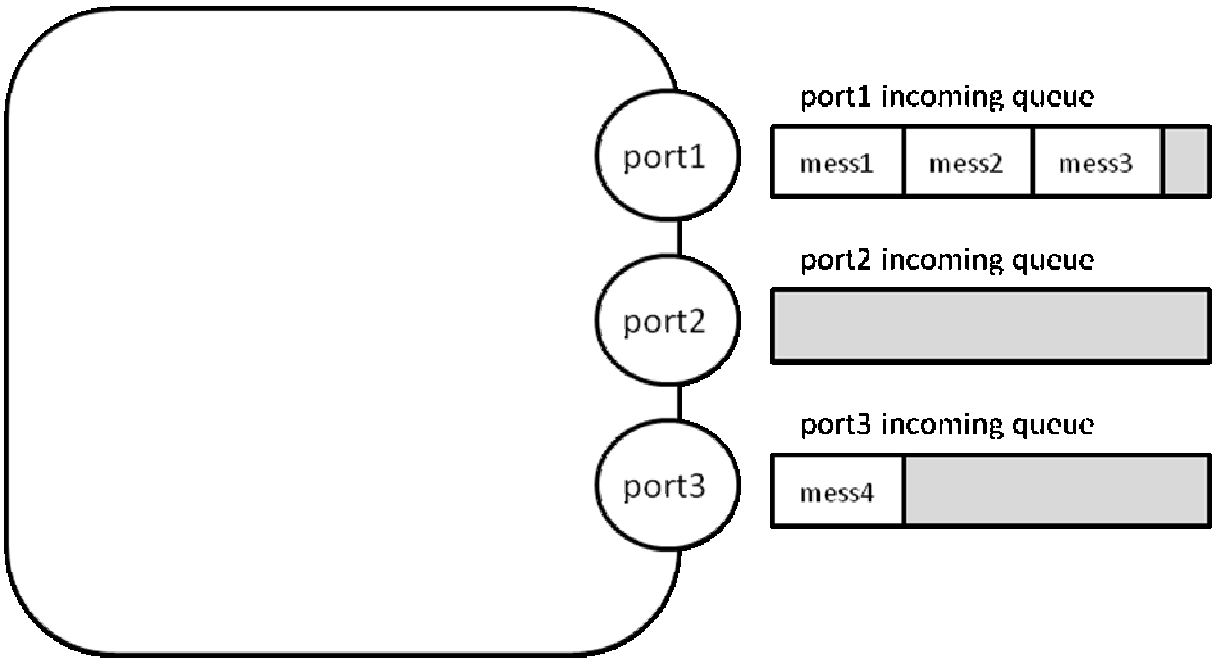
Firstly, because of blocking reading on reception queue, the alt statement can stay blocked because of waiting a message on a specific port queue.

Secondly, if we receive m1 on port p1 and m2 on port p2 at the same time just before testing the branch with p2, this branch will be selected instead of the first branch.



Solution 2: Create an additional queue

A global queue is created for the alt statement. This queue will store messages from every relevant port, with the name of the reception port as additional information.



This TTCN component has three communication ports with their own reception queue. If the alt statement contains tests on port1, port2 and port3, a global queue is created for all three ports.

Every test on reception is done on this global queue.

With this solution, this unique queue is updated only before entering the alt statement or after an alternative branch has been selected. That execution semantic is the one of the snapshot concept.

Problems: This choice of implementation raises two problems: First, we should have access not only to the first message of the queue, but to all messages in the global queue in order to have access to the first message of each port. For example, if the first statement in the alt is *port3.receive(mess4)*, the fourth message should be read and removed from the global queue without touching the others. Second, after a branch of the alt statement has been chosen, the global queue has to be deleted, and the remaining messages have to be resent to their respective queue.

Solution 3: Blocking write-access on port

When entering the alt statement, write access on every relevant port are blocked. By this way, messages in each port will be the same during the entire alt statement execution. Write access will be released after an alternative branch has been selected, or when exiting the alt statement.

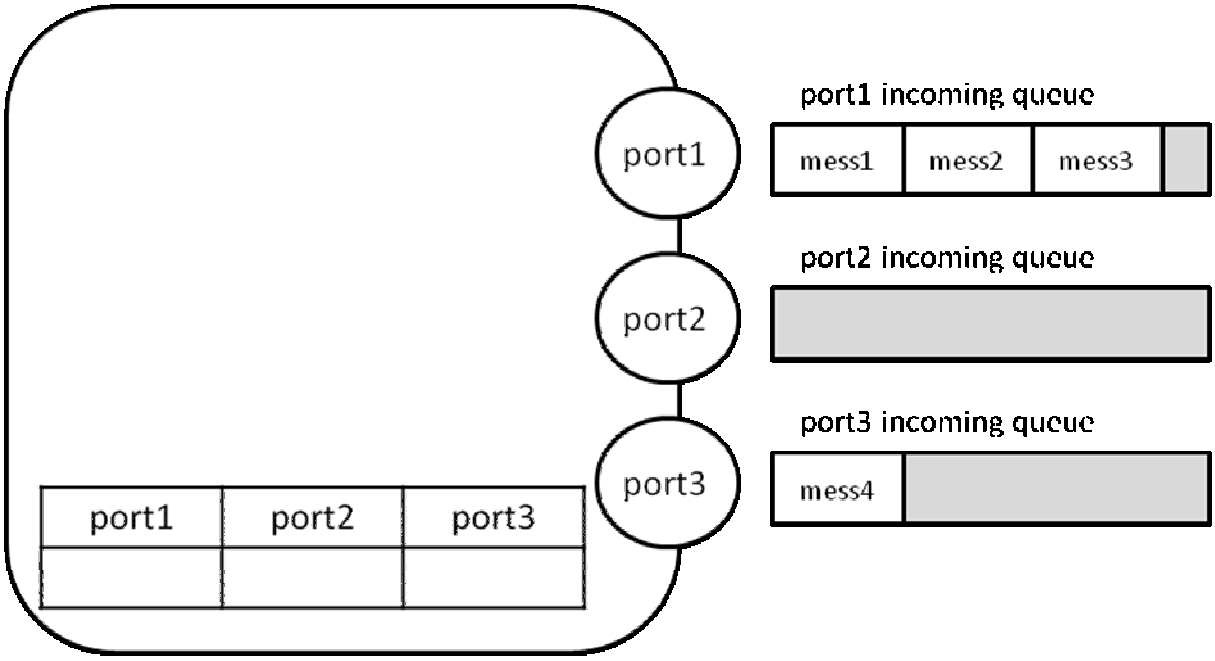
Problem: Blocking write access might suspend the execution of the sender and change its behavior.

Solution 4: Create an array to store the first message of each port

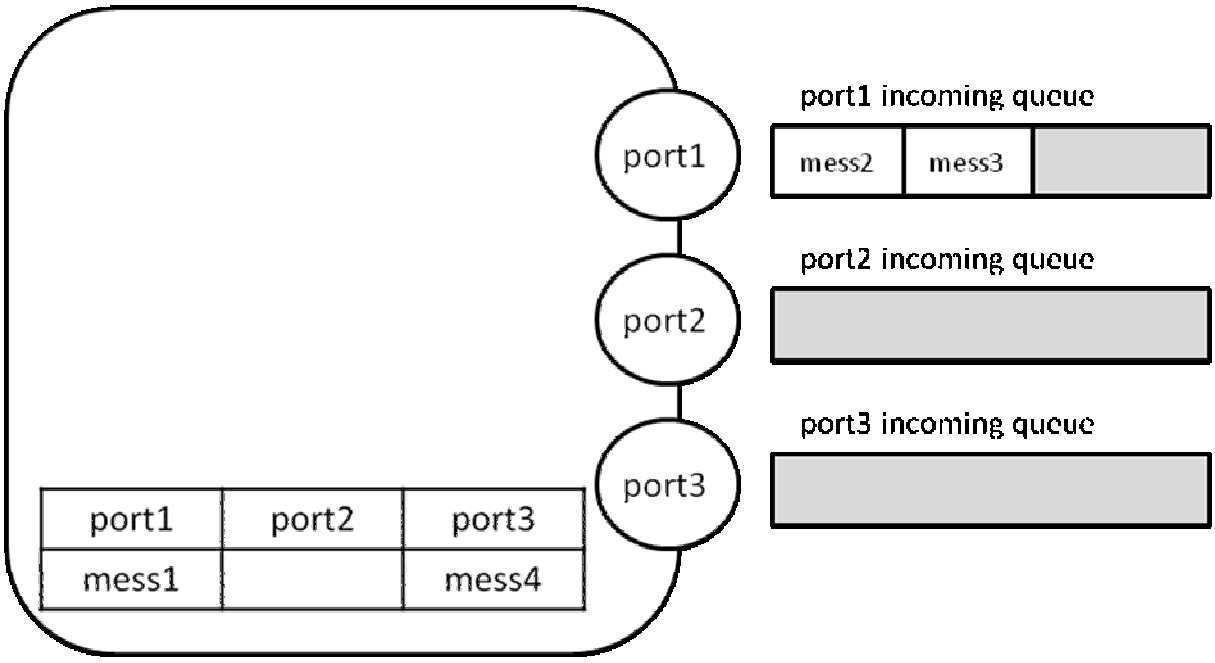
For each component, an array is created. This array will be used to store the first message of each port of the current component. In that case, taking a snapshot will consist in storing the first message (if any) of each port in this array.

During the alt statement, only messages in the array are used for branch evaluation. If an alternative branch is selected, the corresponding message is read (and deleted) and the array is updated. Updating the array amounts to fill every empty field with the first message (if any) of the corresponding port.

Using this solution, the array will not only be used during the alt statement, but also for any communication statement. This means the array should be updated after each **receive**, **getcall**, **getreply** operation (which are actually alt statement with only one alternative branch).

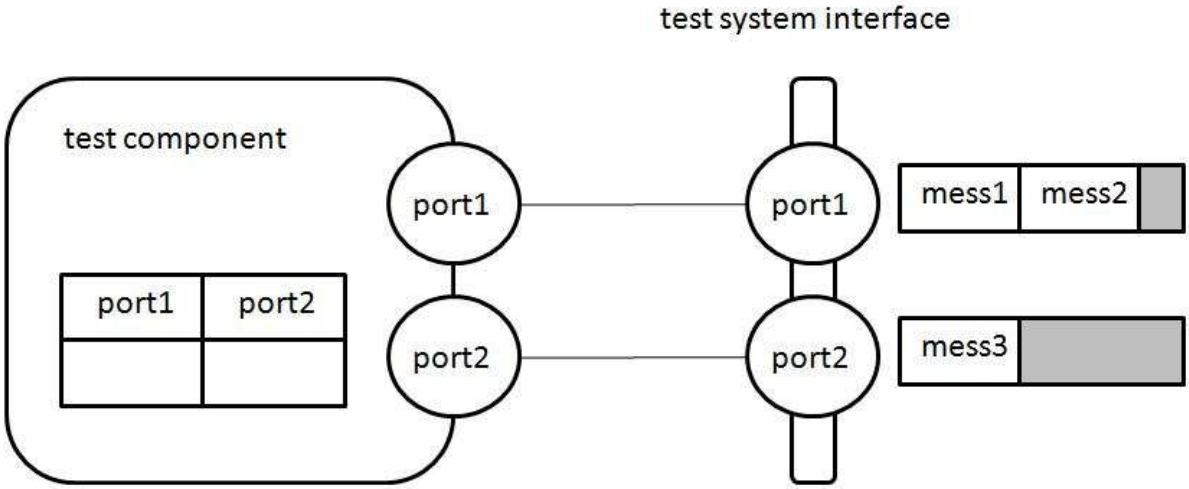


This TTCN-3 component has three communication ports. Each port has an incoming queue which stores every received message. An array is created with a field for each port.

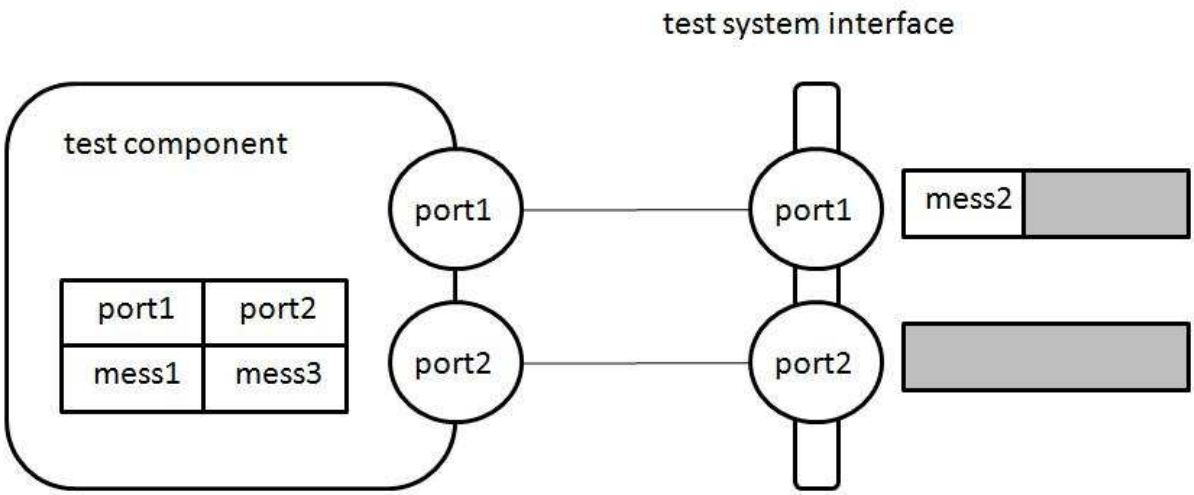


After update, first message of port1 and port3 are stored in the array, and alt statement will only have access to these messages.

Problem: If the system interface has several ports, and at least one message is pending in two of them, and that these ports are mapped to the same component:



After a receive operation in the test component, the first message of each mapped port will be stored in the local array:



Messages coming from system interface are not specific to the current test case. When the test component will die, mess3 will be lost. In the next test case, if a component waits for mess3, it will never receive it whereas mess3 was present.

Conclusion

Test case behavior is a suite of operations, including communication operations that are blocking operations. These operations can be express sequentially, but it can also be possible that the system under test might behave in different ways. This is for this situation that the alternative statement is used. Unfortunately, alt statement and snapshot concept are not easily implementable and usable in major RTOS.

Several solutions have been considered to implement these concepts, each one with their pros and cons. Finally, to implement the TTCN-3 alternative and snapshot concepts in our tool, the choice has been to create an array for each component. This array will be the snapshot of the system. This solution is the closest to the TTCN semantics.