Towards More Accurate Real Time Testing

Prof. Richard O. Sinnott
National e-Science Centre
University of Glasgow, Glasgow G12 8QQ
ros@dcs.gla.ac.uk

Abstract

The languages Message Sequence Charts (MSC) [1], System Design Language[1] (SDL) [2] and Testing and Test Control Notation Testing[2] (TTCN-3) [3] have been developed for the design, modelling and testing of complex software systems. These languages have been developed to complement one another in the software development process. Each of these languages has features for describing, analysing or testing the real time properties of systems. Robust toolsets exist which provide integrated environments for the design, analysis and testing of systems, and it is claimed, for the complete development of real time systems. It was shown in [4] however, that there are fundamental problems with the SDL language and its associated tools for modelling and reasoning about real time systems. In this paper we present the limitations of TTCN-3 and propose recommendations which help minimise the timing inaccuracies that would otherwise occur in using the language directly.

1. Introduction

MSC, SDL and TTCN-3 are international standards used for capturing requirements, for design and analysis, and testing of systems respectively. The languages themselves have been developed explicitly to work together and toolsets are available supporting, at least in principle, a complete environment for requirements capture, design/analysis and testing of systems.

MSC is an international standard that provides a graphical 2-D language often used for the capture of requirements through description of interaction scenarios. Often these scenarios are used to define the properties (traces) that a given system should satisfy[3]. The MSC standard is maintained and updated every four years by ITU-T. Similarly, SDL is an international standard updated every four years by ITU-T. SDL supports the specification of complex software systems and has been extensively applied across a broad array of domains from telecommunications, automotive through to general software development. SDL and MSC have evolved over an extended period of time and advanced toolsets now exist that allow for the automatic verification of MSC against SDL models.

TTCN-3 has been put forward by ETSI and is the only internationally standardised testing language. TTCN-3 provides numerous structures that facilitate a broader applicability than for example was the case with earlier versions of the TTCN standard – which was primarily focused upon conformance testing of OSI protocols. Toolsets are now available for TTCN-3 such as [5,6,7,8].

MSC, SDL and TTCN-3 all have features for describing real time properties of systems. The need to harmonise the real time features of these languages is thus essential to ensure that real time requirements can be expressed and shown to be satisfied (or not) by given models, and ideally that tests that might be generated from those models capture both the functional and real time information needed to test such systems. Similarly, changes in design or requirements, or new information established about the real time properties of the system being tested should be fed into the models, designs or tests to investigate their impact. For example, real time requirements might not be realistic when testing in a live environment is undertaken, or it might be the case that a given model has timing properties that can be shown to cause erroneous behaviours in under certain conditions. Tractable real time between the languages and supported by the tool sets is thus highly desirable.

Diagrammatically the inter-relationship between these three languages is depicted in Figure 1. In this paper we focus in particular on the language TTCN-3. To accurately capture the real time properties of a given test system and to ensure that requirements and models are used to generate realistic and enforceable timing information, it is essential that the language itself (TTCN-3) has a well understood and semantically sound model of time.

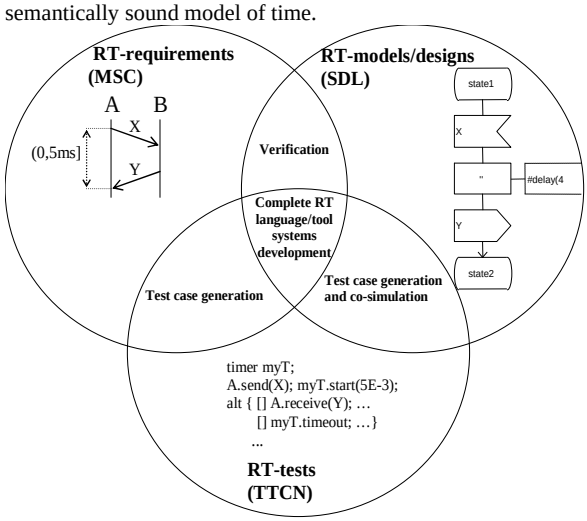

Figure 1: Relation Between MSC, SDL and TTCN

We note that testing is in general a difficult activity, but one that is fundamental to relate implementations to specifications (and vice versa). Real time testing is even more difficult to achieve. To understand why this should be the case, consider the following pseudo-code in Figure 2.

```
SomeFunction()
 { Switch (…){
    Case 1(…)   on average takes T1s
    Case 2(…)   on average takes T2s
    Case 3(…)   on average takes T3s
 }}
```

Figure 2: Real Time Pseudo-Code

If we need a reliable real time system then we need to understand that each of these *Case* statements will take some time. In testing a system containing (perhaps internally) such a function call, several possibilities exist. For example, the average time taken by the *SomeFunction()* operation might be used. However, if *SomeFunction()* was doing some sorting of a file, then the length of time needed to sort this file would be dependent upon the size of the file, whether it had indexes

[1] Formerly known as Specification and Description Language.
[2] Formerly known as Tree and Tabular Combined Notation.
[3] Depending on the semantics, MSC can be used to capture traces that should always be satisfied, never be satisfied or that there exists at least one system execution which can generate the trace.

already built etc. In short, the context in which this function was to be called needs to be established. Note that in engineering real time systems, to ensure reliability, it is often the case that such choices of functionality are specifically engineered to have similar timing constraints.

2 Background to TTCN-3

TTCN-3 is a flexible language applicable to the specification of all types of reactive system tests over a variety of communication interfaces. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing, module testing, API testing etc. Unlike earlier version of TTCN which were primarily focused upon conformance testing, TTCN-3 can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing, as well of course as conformance testing.

From a syntactical point of view TTCN-3 is very different from earlier versions of the language as defined in ISO/IEC 9646-3 [9]. However, much of the basic functionality of TTCN has been retained, and in some cases enhanced. TTCN-3 includes a core notation which is textual in nature (and used throughout this paper), and numerous presentation formats. For example, there are presentation format based on a tabular representation (as per TTCN, TTCN-2) [17], and a graphical MSC-like notation [18].

The TTCN-3 language itself includes the following characteristics:

- the ability to specify dynamic concurrent testing configurations;
- operations for procedure-based and message-based communication;
- the ability to specify encoding information and other attributes (including user extensibility);
- the ability to specify data and signature templates with powerful matching mechanisms;
- type and value parameterization;
- the assignment and handling of test verdicts;
- test suite parameterization and test case selection mechanisms;
- combined use of TTCN-3 with ASN.1 [10] (and potential use with other data typing languages);
- well-defined syntax, interchange format and static semantics;

TTCN-3 also offers a precise operational semantics [11]. Through TTCN-3, abstract test suites can be developed which offer a generic, i.e. primarily implementation independent, way for expressing the functional tests for arbitrary systems/implementations.

These abstract test suites will then typically be compiled into executable test suites, potentially including other information needed for the specific system under test. In addition to the TTCN-3 language description and semantics itself, the TTCN-3 standards included the description of a Test Runtime Interface [12] and a Test Control Interface [13]. Conceptually, a TTCN-3 test system can be thought of as a set of interacting entities where each entity corresponds to a particular aspect of functionality in a test system implementation. These entities manage test execution, interpreting or executing compiled TTCN-3 code, realize proper communication with the system under test, implement external functions, support functions for the logging of information on the test execution itself, and handling of timer operations.

We note that TTCN-3, whilst being a major enhancement to earlier versions of TTCN-2 has the same features (with the same semantics) for describing, testing and evaluating real time aspects of systems. These are based upon the concept of the timer and the notion of the snapshot semantics.

3 Real Time Features of TTCN-3

The TTCN-3 language has certain features which can be applied for the testing of real time systems. The central feature used in TTCN-3 for dealing with timing aspect of a given system under test is the *timer*. TTCN-3 has operations to start, stop, read or check if a timer is running as shown in Figure 3.

```
aTimer1.start;      aTimer2.start(2E-3);
            //timer values of type float
aTimer1.stop;       all timer.stop;
            //stopping inactive timer has no effect
var float aVar;     aVar := aTimer.read;
            //assign to aVar time elapsed since aTimer started
if (aTimer.running) {…}
            //returns true / false if timer is running or not
```

Figure 3: Timers in TTCN-3

Timers are declared, started and either stopped or timeout. When a timer times out, e.g. *aTimer.timeout,* the expired timer is placed in a timeout list. This list is checked when the next snapshot (described below) is taken. We note that there are no global timers in TTCN-3 (nor global data), that is timers may only exist in functions, test cases or in the control part of a test suite. Examples of timer usage in TTCN-3 are shown in Figure 4.

```
timer T1 := 10;
execute testCase1();
T1.start;
T1.timeout;              // pause before executing next test case
execute testCase2();

execute (testCase3(), 5E-3) -> returnVal;
   // returnVal type verdictType = error if result not returned in 5ms

P.call(X,5E-3);        // can also put timeout value on procedure call
      { [ ] P.getreply(X);
        [ ] P.catch(timeout); { verdict.set(fail); … }
      }

while (T.running or x<10) { execute testCase4(); x:= x + 1; }
```

Figure 4: Timer Usage in TTCN-3

3.1 Snapshot Semantics

TTCN-3 has same snapshot semantics as TTCN-2. That is, when a message is sent from a test system to a system or implementation under test and a set of alternative alternatives are possible, the possible responses (messages from the system under test to the test system) are processed in their order of appearance. Each time around a set of alternatives a snapshot is taken of which events have been received and which timeouts have fired. Only those identified in the snapshot can match on the next cycle through the alternatives. This “snapshot” taking effectively freezes the environment whilst alternatives are processed, and if no match occurs, a new snapshot is taken. Whilst supporting consistent testing, this approach causes problems when real-time testing is needed.

3.2 Key Issues with TTCN-3 for Real Time Testing

There are several key problems with current approaches for real time testing with TTCN-3 and the usage of timers. These are: the speed of the tester and the associated problems of the snapshot semantics and its impacts on accuracy of

timing information as well as the need to allow for abstraction from test hardware/software properties; dealing with time critical testing information; the problems of time synchronisation of distributed test configurations.

3.2.1 Problems with Snapshot Semantics on Speed of Tester

The simple timer concept is intended to catch long-term timeouts and as such is not powerful enough to deal with hard real-time. Since the standard snapshot semantics may summarize time-critical events into one snapshot, important timing or ordering information might get lost. This is in contrast to the TTCN standard which states "*a Test Case or Test Step should not contain behaviour where the relative processing speed of the MOT (Means of Testing) could impact the results*". In this case, it is not decidable, whether a violation of real-time constraints occurred or not. We note that exactly such behaviour is needed to describe real-time test cases. The verdict itself will rather depend on the question of how the alternatives are ordered in the TTCN behaviour description.

To overcome this, the TTCN-3 semantics assumes that taking a snapshot is instantaneously, i.e. has no duration. The TTCN-3 standard also describes the evaluation of a snapshot as a series of indivisible actions of a test component. It states that: *the semantics do not assume that the evaluation of a snapshot has no duration. During the evaluation of a snapshot, test components may stop, timers may timeout and new messages, calls, replies or exceptions may enter the port queues of the component However, these events do not change the actual snapshot and thus, are not considered for the snapshot evaluation*. As we will show, this causes errors in the accuracy of real time testing.

It is a fact that if the test executor equipment (the means of testing) is not fast enough, it can not communicate with the implementation under test properly so the testing becomes meaningless. The most important advantage of reactive testing in contrast to log analysis is that the tester can explicitly control the behaviour and state transitions of the implementation under test in order to achieve better coverage. The performance bottleneck, for example, can be the delay between two successive send events or the reaction time (the time between receiving a message and sending the response). It is necessary to determine whether a means of testing is fast enough for a certain test case.

3.2.2 Problems with Assignment of Test Verdicts

In TTCN-2 a verdict of a test could only have three values PASS, FAIL and INCONCLUSIVE without further qualification. TTCN-3 extends these values to allow for ERROR and NONE values also. However, when it comes to time critical measurements, it may be necessary to exploit more information. For example, if an event has to happen within a time interval, it may be interesting or important to know how well it is within the interval, e.g. the time when an event happens has to be recorded in the test report.

3.2.3 Problems with Time Synchronisation of Distributed Test Configurations

A test system itself may consist of many Parallel Test Components (PTCs) and a main test component (MTC). In order to be able to test for timed behaviour spread over different PTCs, a time synchronisation mechanism is necessary, i.e. each PTC has to be able to ask for the exact absolute time at any moment in order to relate test events to this time.

There are several ways in which these problems can be addressed. One way to address or at least minimise these problems is through general guidelines on the usage of TTCN-3 for real time testing. Another more direct way is through language enhancements that specifically address these problems. Currently no language enhancements have been made completely overcoming the problems in timing caused by the snapshot semantics.

4 Using TTCN-3 for Real Time Testing

We note that the language usage aspects presented here are given in TTCN-3 syntax but the general principles presented also apply to TTCN-2 usage for real time systems testing. We begin with a summary of the existing features of TTCN-3 that can be used for real time testing.

With regard to these first of the bullet points and section 3.2.1 given above, the most important question that should be asked for real time testing is, is the tester fast enough to test the specific implementation under test? That is, is the time needed to send/receive messages, access/modify ports/timeout lists etc much less than real time properties of implementation under test to be checked, i.e. negligible? If the test system is too slow then real time testing for this implementation under test with this test system is not possible. However, even in the case where the test system is fast enough, the tester should be aware of the impacts of the speed of the test system on the test results.

As an example of this consider the following scenario where we want to check that a tester sending *X* to an implementation under test can result in *A, B* or *C* being received. Further suppose we want to check that if the implementation under test responds with *C* in *<=0.05ms* then the tester should send *Y* otherwise if the implementation under test does not respond before *0.1ms* then the tester should send *Z*. The simplified MSC 2000 for this scenario is shown in Figure 5.

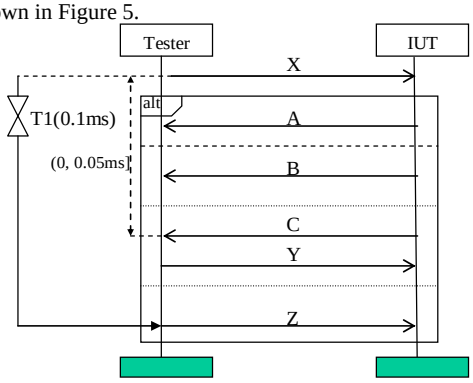

Figure 5: MSC 2000 Scenario Representation

The corresponding TTCN-3 representation of this scenario is depicted in Figure 6.

```
var float t; timer T1:=0.1ms;
Tester.send(X); T1.start;
alt {[] Tester.receive(A); ….
     [] Tester.receive(B); ….
     [] Tester.receive(C); t:=T1.read;
          if(t <=0.05) { Tester.send(Y);…} else {...}
     [] T1.timeout; { Tester.send(Z); … } ….
}
```

Figure 6: TTCN-3 Scenario Representation

At first glance there does not appear to be any problems with this representation. However, for successfully performing the real time test, certain information is not present. For example, tester timing issues are not given, e.g.

how long does it take to: start a timer; take a snapshot; check a port and try to match alternative; check a timeout list; do an assignment; check a Boolean expression etc? In short the real time test should be based on the real time properties of the implementation under test being tested and not dependent upon the speed of the test system itself. Hence information regarding the speed of the test system itself has to be provided.

In addition, the previous scenario states nothing about the transit delays associated with the testing, e.g. the time taken for the network transit delay in sending/receiving is not specified, or depending on the implementation under test's configuration with the test system, the time taken for messages to travel through lower protocol layers through which the implementation under test is accessible. We note that this is not really in description of the requirements, but it should be, if want accurate real time measurements, i.e. accurate real time tests of the implementation under test.

A further problem with this scenario representation is the likelihood of inaccurate time measurements caused by the snapshot semantics. That is, as a result of the existing TTCN snapshot semantics, the actual value of the timeout is incorrect. Specifically, *Z* is sent at time equal to the time of the last snapshot plus the time to check for (match alternatives) *A/B/C* plus the time to check the timeout list itself. This additional delay in sending *Z* could be important! The value recorded in the reading of the timer (*T1.read*) is also affected. The TTCN-3 standard currently states that "*the read operation is used to retrieve the time that has elapsed since the specified timer was started and to store it into the specified variable*" which implies that this value is read there and then, i.e. its current value. However, there are situations for example in a Boolean guard on an alternative statement where the value of the read will be the value of the timer when the last snapshot was taken. This latter case, addresses the possible difficulties in Figure 7.

```
timer T1:=1s;
…
alt {[T1.read>1] Tester.receive(A); ….
     [T1.read<=1] Tester.receive(B); ….
     [] T1.timeout; ….  }
```

Figure 7: Potential Timing Problems

Here it is possible that the first two qualifiers are never satisfied since the first *T1.read* alternative might be false, then by the time the second alternative is evaluated, i.e. after the time to access the *T1* value again, the second qualifier might evaluate to false also, which would of course be especially counterintuitive. Whilst making the test case decidable, it is clear that the actual value of the timer that will be used in evaluating these Boolean guards - as prescribed by the standard! - will be wrong. Specifically, it will be earlier than the current real time (since it records the value of time at the previous snapshot).

5 Guidelines on TTCN-3 Usage for More Accurate Real Time Testing

To address these problems we produce general guidelines on TTCN-3 usage for real time testing. Through these guidelines specific timing aspects of the tester (tester benchmarking) should be taken into account. Further proposals on language usage to aid in addressing the timing critical aspects related to transit delays and recording of timing information are illustrated. Finally examples of language usage which help improve test component synchronisation are presented.

5.1 Guidelines to Establish the Speed of the Tester

The basic idea behind this proposal to establish the speed of the tester is based upon exploiting the TTCN-3 language itself to provide some form of minimum benchmarking of the test system itself. For example, to test the real time aspects of a particular implementation under test it might be necessary to establish that the tester is capable of performing a particular number of sends in a given time period. Establishing this form of additional information can be achieved through specific functions represented in TTCN-3 which should be called before the real time tests of the implementation under test itself are made. We note that such additional information is typically provided in TTCN-2 test suites via a Protocol Implementation Conformance Statement (PICS) or Protocol Implementation Extra Information for Testing (PIXIT) statement [19]. These contain additional information such as the physical set up and connection of the test system, e.g. test hardware description. An example of one such function call is given in Figure 8.

```
module testExec (float maxExecTime )
{  // definitions of data types, test data templates,  ports, etc
function testExecutor() runs on myMTCType return verdicttype
{ var float timeTaken;
  timer T:= 5*maxExecTime; T.start;
              // note timer includes a scaling factor
  P1.send(aType);
             // or more complex behaviour with
             // multiple sends/assignments
  timeTaken := T.read;
  if (timeTaken>maxExecTime) { return fail;}
             // could use timeout if no scaling factor
  else { return pass;} };
…
control {var verdicttype execVerdict := pass;
   execVerdict := testExecutor();
     if (execVerdict == pass)
        { execute(theRealTimeTestCasesNotSpecifiedHere()) }
     else {myMTCType.stop}}
```

Figure 8: Tester Benchmarking via TTCN-3 Functions

Here the test suite parameter *maxExecTime* has to be provided and is subsequently used to ensure that the test system is fast enough, i.e. function *testExecutor()* evaluates to *pass*. Other possibilities such as lower timing bounds, or parameters to deal with system load or performance more generally, can also be used to parameterise a test suite and subsequently used in function calls to evaluate whether the test system is suitable for testing the given implementation under test. In addition to testing that the test system is capable of performing a sufficient number of sends within some upper time limit, more complex interaction scenarios to establish the speed of the tester can also be established. For example, it is possible to perform benchmarking on sends and receives of individual components as shown in Figure 9.

```
function testSpeedofTester ( ) runs on myPTCType return  integer
  {var address thePTCid := self;
   var integer c:=0;  timer T:= 10;
   connect(self:P1, self:P2);
   T.start;
   while(T.running)
    { P1.send(someType: * ) to thePTCid;
      P2.receive(someType: * ) from thePTCid;
      c := c + 1;   // other time consuming behaviour here?
    }return c;
  };   // then in control part should make sure that c >= some value
```

Figure 9: Extended Tester Benchmarking

Through this approach of connecting the tester to itself (*connect(self:P1,self:P2)*) and sending representative messages and associated data sets, details regarding the

timing for performing sends/receives, taking snapshots, performing assignments etc can be obtained. In short, it is possible to obtain precise measurements about the speed of the tester which can subsequently be used in evaluating the real time behaviour of the implementation under test. Thus for example, the time taken to perform sends/receives etc would typically be subtracted from the response times from the implementation under test.

5.2 Guidelines on Establishing Time Critical Information

It is often the case that additional timing information on verdicts than just *PASS, FAIL, INCONC, ERROR, NONE* are required. This is especially the case when testing real time systems where for example it is often necessary to establish the time inside/outside some threshold or the time taken to access the implementation under test itself. Consider the MSC 2000 interaction scenario given in Figure 10.

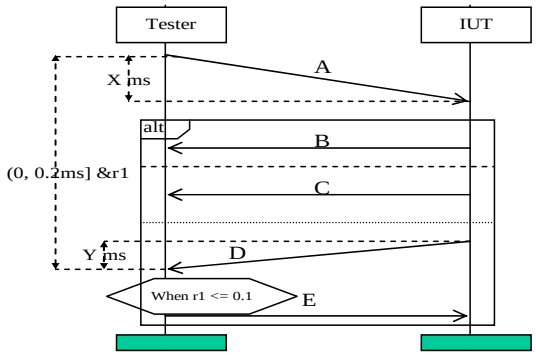

Figure 10: MSC 2000 Scenario With Transit Delays

Here the requirement is that the transit delay between the sending of message *A* from the tester to the implementation under test should be *Xms* and the transit delay due to message *D* being sent from the implementation under test to the tester should be *Yms*. This MSC interaction scenario also requires that if the value of the time between *A*'s consumption[4] by the implementation under test and *D*'s consumption at the tester is less than *0.1ms* then message *E* should be sent. This can be represented through the TTCN-3 depicted in Figure 11.

```
timer T1:= 0.2ms-Xms-Yms;
Tester.send(A); T1.start;
alt { [] Tester.receive(B); …
      [] Tester.receive(C); …
      [] Tester.receive(D);
          {r1 := T1.read;
           if (r1-Xms-Yms <=0.1ms) { Tester.send(E); ...
              }…
           else { // do something else here not specified here
                }
       ... }
       [] T1.timeout;…}
```

Figure 11: TTCN-3 Transit Delay Representation

Here it should be noted that the necessary transit delays incurred can be represented directly through inclusion of the necessary offsets in the setting of the timer *T1*. We also note that if time critical information such as "the amount of time message *D* was received inside of the *r1-Xms-Yms* limit" can be extracted simply through reading the time associated with the *T1* timer and storing its value in a test case variable (of type float), e.g. by including a statement such as *varT:=T1.read* before the *Tester.send(E)* event in Figure 11.

Hence to summarise, the TTCN-3 language is expressive enough to allow to express transit delays and to record time inside thresholds etc. Provided this information can be established in some way, the language itself can be used to express them.

5.3 Guidelines on Time Synchronisation between Distributed Test Configurations

One existing problem with using TTCN for testing distributed real time systems is with the need to perform timed synchronisation between distributed test components. As stated previously, TTCN-3 does not allow for global test data or for global timers. There are two main ways in which time synchronisation can be addressed within TTCN-3. Firstly either the language itself can be extended with the necessary features to support different synchronisation possibilities. Alternatively, guidelines can be produced which show how the problem related to time synchronisation can be minimised. To understand the problem we consider a distributed test architecture based on concurrent TTCN as depicted in Figure 12.

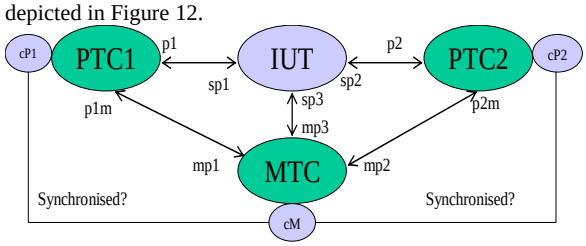

Figure 12: Concurrent TTCN Test System Architecture

Here the implementation under test is to be tested by a test system comprising two parallel test components (*PTC1* and *PTC2*) and a main test component (*MTC*). These test components have the necessary coordination points defined[5] and are thus able to communicate with one another. The components all have their own individual clocks used, e.g. cp1 is the clock of *PTC1*, *cP2* is the clock of *PTC2* and *cM* is the clock of *MTC*.

In addition, this test system requires that to test the implementation under test successfully, the test components (*PTC1, PTC2* and *MTC*) need to be synchronised with one another.

Improved time synchronisation for this test system can be achieved through guidelines on the usage of co-ordination messages for the test components. That is, it is possible to support a certain level of time synchronisation through restrictions on the ports used for transferring time critical information, e.g. messages used for synchronisation of the separate clocks. As an example of this consider Figure 13.

```
function synchronisePTCCheck() runs on PTC1Type
 {timer T;
  var address theSysid := system; var address theMTCid := mtc;
  T2.start(10);
  PCO1.send(myType: * ) to theSysid;
     alt {[] P1M.receive(float: * ) from theMTCid  -> value newTimerValue
                                { T.stop; T:= newTimerValue; T.start;  … }
          [T.read<=5] P1.receive(aType: *) from theSysid { … }
          [T.read>5] P1.receive(anotherType: *) from theSysid { … }
          … // timeout case and other behaviour
 } }
```

Figure 13: Guidelines on Improved Time Synchronisation

[4] Note that MSC does not have the notion of buffering as with other languages such as SDL. A message sent between two instances corresponds to two events: the sending of the message by a producer and the consumption of the message by a consumer.

[5] although we note that co-ordination points and points of control and observation are both represented as ports in TTCN-3.

Here we use an explicit port (*P1M*) which is only used for time updates to avoid problem of queuing timing messages. These time-critical ports are then placed at the highest level of alternative, i.e. they will be evaluated first when the next snapshot is taken. In this example, the first alternative (at port *P1M*) allows for co-ordination messages to update clock (timer) used for synchronising tests. We note here that this function assumes negligible time for the sending, transit delay, port access, stopping of timer, assignment of value, starting of timer associated with the receive event. However, if knowledge of the latency caused by these things is known, e.g. based on some form of tester benchmarking, then appropriate offsets can be used when setting the associated timer.

We note here that this guideline only offers a general improvement for time synchronisation aspects and is not the general solution to the problem. Thus for example this guideline still suffers from the existing snapshot semantics problem, i.e. the co-ordination message used for synchronising the clock is "assumed" to arrive instantaneously at the time of the next snapshot. In reality of course, the message might arrive immediately after a given snapshot and hence will be delayed by however long it takes to evaluate the other alternatives in this snapshot; to take the next snapshot and finally to match on the first clock synchronisation event.

6. Conclusions

This paper has shown the limitations of the existing treatment of time within TTCN-3. The language does allow for the specification of real time tests however it is clear that *real time testing will always be wrong due to the existing snapshot semantics*. It is of course, possible to try to enhance the language to overcome the issues caused by the snapshot semantics. This would be a radical change however to the language and one that would break the existing tools, cause the language to be much more complex to use, and potentially allow for tests to be developed that were inconsistent depending upon the real time performance of the environment of both the test system and implementation under test. In short, it is unlikely that such a radical change would ever be fully accepted. A far better approach is to support general guidelines which, whilst not giving absolute accuracy to tests, will at least give some clear ideas of the boundaries for the level of accuracy in real time testing, and hence a more accurate measurement of the real time!

We note that other approaches have been presented which offer a less radical proposal to enhancements to the language [14,15]. These approaches rely upon time-stamping and logging of timed events. Whilst addressing some of the challenges in real time testing, this approach is still limited by the snapshot semantics of TTCN. It also assumes that the test execution system is fast enough to test the desired real-time properties of the system under test.

One of the problems in using guidelines instead of having real time features correctly realised within the language, is the difficulties this imposes when relationships between the MSC, SDL and TTCN-3 languages are considered. Thus whilst it is possible to generate tests in numerous ways as described in [16], real time information generation will typically not be supported. It is worth noticing that the automatic generation of TTCN tests from SDL models typically does not include real time information anyway. The relationships between the languages as defined in the standards are ill-defined and often left to tool vendors to decide (in how they realise/implement their toolsets [5]). For example, tool vendors often offer choices when dealing with MSC and TTCN timers and how they should be interpreted in relation to SDL. Typical choices include options that allow to: ignore such timing information; if a timer event exists in MSC/TTCN then a matching timer event must exist in the explored SDL path, but a timer event in the explored SDL path is accepted even if there is no corresponding timer event in the MSC/TTCN; or alternatively all MSC/TTCN timer events must match a corresponding timer event in the explored SDL path, and vice versa, where here a corresponding timer event implies a timer with the same name and same timeout value. Such confusion will of course adversely influence the wider uptake of these languages and the general goal of a complete tool-supported environment for the requirements capture, design, analysis and subsequent testing of real time systems.

7. References

[1] ITU-T, Rec. Z.120, Message Sequence Charts (MSC), MSC 2000, Geneva Switzerland.

[2] ITU-T, Rec. Z.100, Specification and Description Language (SDL), SDL 2000, Geneva Switzerland.

[3] ETSI - Methods for Testing and Specification (MTS): The Tree and Tabular Combined Notation version 3, DES/MTS-00063-1 v1.0.9, Sophia Antipolis, France.

[4] R.O. Sinnott, *The Formal, Tool Supported Development of Real Time Systems*, Proceedings of International Conference on Software Engineering and Formal Methods, 26-30 September 2004, Beijing, China.

[5] Telelogic toolset, http://www.telelogic.com

[6] Danet TTCN Toolbox – TTCN-3 toolbox, http://www.danet.de/index.php?id=425&L=3, 2004.

[7] Testing Technologies, www.testingtech.de

[8] Da Vinci Communications Terzo tools www.davinci-communications.com/products, 2002.

[9] ISO/IEC 9646-3: Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular combined Notation (TTCN).

[10]ITU-T Recommendation X.680: Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation.

[11]ETSI ES 201 873-4 V2.2.1 (2003-02), Methods for Testing and Specification (MTS) - The Testing and Test Control Notation version 3, Part 4: TTCN-3 Operational Semantics.

[12]ETSI ES 201 873-5 V1.1.1 (2003-02), Methods for Testing and Specification (MTS) - The Testing and Test Control Notation version 3, Part 5: TTCN-3 Test Runtime Interface.

[13]ETSI ES 201 873-6 V1.1.1 (2003-02), Methods for Testing and Specification (MTS) - The Testing and Test Control Notation version 3, Part 6: TTCN-3 Test Control Interface.

[14]Z.R. Dai, J. Grabowski, and H. Neukirchen. Timed TTCN-3 – A Real-Time Extension for TTCN-3. In I. Schieferdecker, H. Koenig, and A. Wolisz, editors, Testing of Communicating Systems, volume 14, Berlin, March 2002. Kluwer.

[15]H. Neukirchen, Z.R. Dai, J. Grabowski., *Communication Patterns for Expressing Real-Time Requirements Using MSC and their Application to Testing.* Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems (TestCom2004), Oxford, United Kingdom, March 2004.

[16]R.O. Sinnott, *Architecting Specifications for Automated Test Case Generation*, Proceedings of International Conference on Software Engineering and Formal Methods, pages 24-34, September 2003, Brisbane Australia.

[17]ETSI ES 201 873-2 (V2.2.1): Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT).

[18]ETSI TR 101 873-3 (V1.1.2): Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT).

[19] ISO/IEC 9646-1: Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts.