Challenges of Testing Periodic Messages in Avionics Systems Using TTCN-3

Bernard Stepien and Liam Peyton

University of Ottawa - EECS
800 King Edward Ave Ottawa, ON K1N 6N5, Canada
{Bernard,lpeyton}@eecs.uOttawa.ca

Abstract. The TTCN-3 language was conceived initially for testing telecommunications protocols that consist of sequences of discrete messages between communicating entities. TTCN-3 has a clear model of separation of concerns between an abstract layer, where test behavior is specified, and a concrete layer, where messages are encoded / decoded and sent and received to/from the system under test. This model, however, is cumbersome for testing protocols with periodic messages as used in avionics systems. This paper presents an innovative approach to addressing issues involving periodic messages in TTCN-3, based on our experiences working with avionics systems. Extensions to the TTCN-3 standard are proposed, based on our approach. We also demonstrate how the approach can be used for test system certification and requirements verification for avionics systems.

Keywords: periodic messages, testing, TTCN-3, avionics.

1 Introduction

TTCN-3 [1] is a language and international standard that was initially conceived specifically for testing telecommunications protocols that consist of complex exchanges of messages that offer both alternative and interleaved behaviors. TTCN-3 has a clear model of separation of concerns between an abstract layer that specifies the test behavior of a System Under Test (SUT) and a concrete layer that encodes and decodes messages that are sent to and received from the SUT. The link between the two layers is an abstract representation of data that enables the use of generic tools not requiring any programming effort from the user to perform the matching of received data with test oracles as shown in Fig. 1. The matching mechanism is a central concept in TTCN-3.

Telecommunications system protocols are usually composed of a limited number of discrete messages. However, avionics systems that use periodic messages send or receive possibly very large series of messages with an identical payload at precise time intervals. In contrast to telecommunications protocols, protocols with periodic messages are mostly based on unacknowledged communication using the UDP protocol in broadcast mode. This is mostly due to the fact that the repetition of the same message will ensure that at least a minimum amount of messages will eventually get through and thus maintain the receiving system in a stable state. The periodicity is also verified by the receiving entity to determine if a message is not obsolete and thus

H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): ICTSS 2013, LNCS 8254, pp. 207–222, 2013.

should be used. Finally, an important feature of periodic messages is that sequence numbers are used to determine if a message is not obsolete since they do not use any confirmation mechanisms. These sequence numbers are the only data element that varies from one message to the next in the otherwise identical payload.

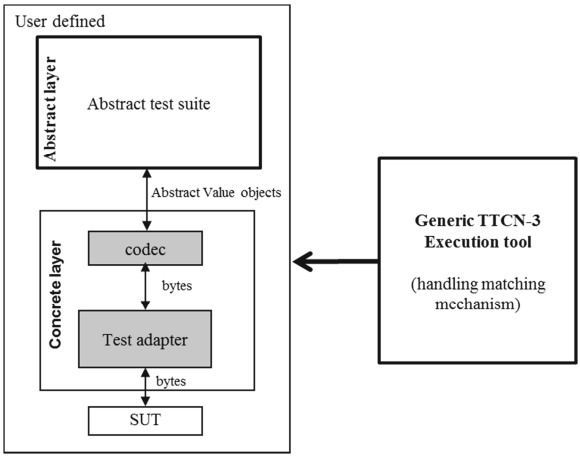

Fig. 1. The TTCN-3 separation of concerns model

This paper presents an innovative approach to addressing issues involving periodic messages in TTCN-3, based on our experiences working with avionics systems. Extensions to the TTCN-3 standard are proposed, based on our approach. We also demonstrate how the approach can be used for test system certification and requirements verification for avionics systems

2 Background

Various extensions to TTCN-3 have already been developed for other application domains. The extension for continuous signals [4] [19] introduces the notions of time, sampling, streams, stream ports, stream variables and the definition of an automaton similar to control flow structure to support the specification of hybrid behavior. The extension for real time requirements [16][17] introduces a test system wide available system clock and means to access the time points of the relevant interaction events between the test system and the system under test (SUT) with precision, namely to avoid delays caused by the test tool's processing. The extension for static test configuration and deployment [14] introduces a concept of special configuration function which can only be called in the control part of a TTCN-3 test suite. This consists in defining static test components that straddle several test case executions and have the result of not resetting timers and message queues. The extension for extended parameterization [17] introduces extended value and type parameterization for the various TTCN-3 parametric language elements. The extension for behavior type [18] allows defining types for

altsteps, functions and test case to render them parametric. The extension for extended TRI [20] introduces the capability to pass abstract values from the abstract to the concrete layer, thus bypassing the codec step as shown in Fig. 1. Early use of TTCN-3 is reported for avionics testing in the context of automated test case generation. An approach to periodic messages using the current version of the TTCN-3 language has been attempted using procedure oriented testing [5]. However, avionics systems use messages based on structured data types, similar to telecommunications messages, for which message oriented testing would seem more appropriate.

3 Periodic Messages and the TTCN-3 Model

3.1 Differences between Discrete and Periodic Message Systems

There are conceptual differences between protocols of systems that use discrete messages and those that use periodic messages. In systems with discrete messages each message is unique and usually only occurs once or a limited number of times. Also, messages usually occur in a given sequence that can be strict as shown in Fig. 2 or follow more complex patterns of interleaving.

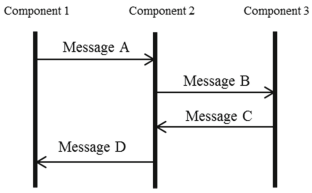

Fig. 2. Discrete message protocol

In systems with periodic messages, such as the ARINC 629 protocol [9], a given message is repeated automatically potentially a large number of times either with identical content or minor differences in content as shown on Fig. 3. Most of these messages are sent to the avionics system by various sensors but less frequent messages are also originating from the avionics equipment side to indicate status. More challenging is the case where heterogeneous messages of different types occur concurrently and over the same communication medium.

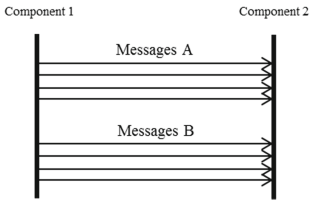

Fig. 3. Periodic messages systems

The typical TTCN-3 separation of concerns requires unnecessary repeated encoding or decoding of identical messages. To avoid this redundancy, we propose a new approach that consists in handling periodicity in the concrete instead of the abstract layer as shown on Fig. 4. Messages are sent only once by the abstract layer until the content needs to be changed and received by the abstract layer only when the data received in the concrete layer has changed. There is a difference between handling messages that are sent and messages that are received. Periodic identical messages being sent need to be encoded only once and are first sent to a scheduler that handles the periodicity. A mechanism updates the data buffer containing the encoded message whenever content changes and controls the scheduler to modify periodicity.

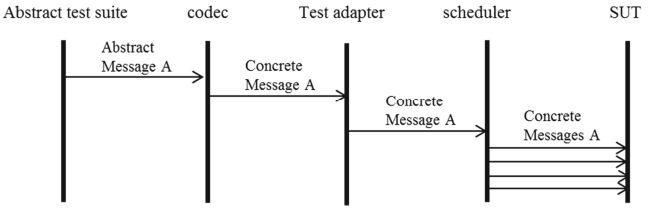

Fig. 4. Separation of concerns for sending periodic messages

Messages being received are handled through traditional asynchronous receiving threads, but, with a major difference that consists in first checking if a newly received message is identical to the previously received message merely by comparing their bytes, i.e. without first decoding the message. If there is a change, the received message is en-queued and presented to the abstract layer for matching with test oracles as shown on Fig. 5. If the content has not changed, the message is discarded.

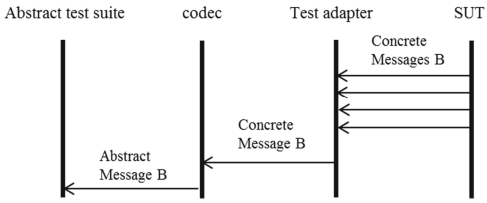

Fig. 5. Separation of concerns for receiving periodic messages

This approach is particularly efficient in the case of concurrent periodic messages of different types which would present enormous interleaving challenges if taken raw. Also, this approach results in the specification of changes of message content that appear in the abstract test suite as if they were discrete messages, thus removing the complexity that concurrent message types would create if their periodicity would be handled in the abstract level.

One of the major features of TTCN-3 is logging. Each message sent or received is logged by the test execution tools and thus enables efficient tracing. Although logging is mostly a feature of TTCN-3 execution tools except for the TTCN-3 explicit log

statement, there is no standardization of logging as such. Logging remains the concern of tool providers. However, since each message is logged, periodic messages inevitably would produce a massive amount of logging making analysis of results tedious if not impossible. Various tools provide log filtering features based on parallel test components, but these would not be usable for periodic messages. Solutions have been found by tool providers that produce logs only when messages have changed. These solutions are not portable from one tool to another from different vendors because there are not covered by the standard and each vendor may choose a different approach. Also, the requirement specific to the avionics industry of using a scheduler [8] makes the handling of periodicity by the abstract layer useless by definition. The implementation at the concrete layer requires dispatching messages arriving through the *triSend*() method to the appropriate message type data buffer that is then used by the scheduler as shown on Fig. 6.

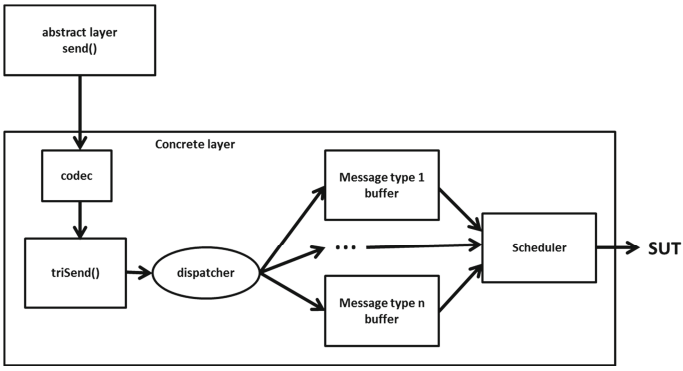

Fig. 6. Concrete layer architecture for periodic messages

In avionics, reducing weight is a prime preoccupation. This includes reducing the needs for equipment of all sorts including networking equipment such as cables. This is achieved by using data concentrators that receive data from nearby sensors or other equipment on a many to one principle and then forward an assembled complex message to the final destination (cockpit) on a single cable. Different message have different periodicity or may even be aperiodic which requires the use of the concept of scheduler that is subject to intense research such as in [11] [22] [23]. It is the scheduler that computes sequence numbers. One of our findings was that the requirement of schedulers naturally eliminates the use of parallel test components that is one of the central concepts of TTCN-3.

3.2 Proposed Changes to the TTCN-3 Language

While a prototype of this model has been fully implemented in the concrete layer with the current version of the TTCN-3 standard and thus using a general purpose object oriented language, some modifications to the TTCN-3 language would provide a more rigorous approach by enabling one to specify the test fully at the abstract level

leaving the implementation details as built-in features in the hands of TTCN-3 tool providers rather than the users, thus preserving semantics across TTCN-3 tools. TTCN-3 is a strongly typed language. For periodic messages, there are a number of enhancements that would allow mostly checking the use of operations that are specific to periodic messages and cannot be used for discrete messages protocols. We list our proposed changes here.

Port Declarations

The port declaration for messages needs an additional keyword periodic to indicate that the messages are periodic as for example.

```
port myPort periodic message {
  inout MyPeriodicMesssageType;
}
```

Also, in the industrial applications we studied, there were cases where both periodic and discrete messages are using the same port. This would lead to a need for a mixed communication mode. The keyword mixed, by itself is already reserved for indicating that the port is used for both message and procedure oriented communication. Here the combination of periodic and mixed keyword could potentially solve the problem.

Data Type Declarations

There are three differences in data typing for periodic messages:

- Specify that the data type is to be used for periodic messages only.
- Specify the periodicity and any changes thereof.
- Handling sequence numbers.

Specifying the Periodic Nature of the Message and Its Periodicity

The first two differences for data typing periodic messages involves specifying the time interval between sending two periodic messages and determining if received messages are received at time intervals that exceed the periodicity threshold in which case a timeout event should be reported to the abstract level. This consists in the capability to specify both an initial or default periodicity and also new periodicities in the course of a test sequence. The initial periodicity would be part of the data type declaration itself rather than a port declaration since a port can be used for different message types that could have different periodicities or even aperiodic messages. For example to specify an initial periodicity (default) of 30 ms we would use the following syntax:

```
type record MyType periodic 0.030 {
   // field definitions
}
```

However, one test purpose could be the variation of the periodicity to trigger faults in the SUT and verify that they occur. For that purpose we also need a new abstract language construct in order to be able to specify modifications of the periodicity in a similar way as TTCN-3 enables modifying the values of timers with a fundamental difference that with timers we use a timer instance while with data types we use the data type name only. This is based on the fact that for periodic messages data types we have only one method that handles this periodic data type in the test adapter. We propose the method name *setperiodicity* with a float value that is applied to a data type name for this purpose as for example:

```
MyPeriodicType.setperiodicity(0.100);
```

Another aspect of periodicity could be that the periodicity would be different whether the messages are sent or received. This issue certainly would arise with concrete telecommunication messages where a given message type is used in both directions but our experience with periodic messages shows that periodic messages have a different data type depending on the direction. Thus, that distinction is not a priority but must be considered as future work. This feature also requires a mirror method in the standard test adapter class: *triSetPeriodicity(Object datatype, double periodicity).*

Handling Sequence Numbers

Sequence numbers must also be handled differently for periodic messages. Normally, each message has a different sequence number whether the payload is the same or not. Since the periodicity is handled in the concrete layer, the scheduler is responsible for assigning the correct sequence numbers. Thus, when creating a template at the abstract layer, the sequence number field should not be instantiated. This is somewhat similar to the treatment of optional fields. This means that a sequence number field can be omitted without producing a compile time error as for mandatory fields. Here the field is merely ignored in the abstract layer but we still need to declare what the initial and maximal values that the concrete layer should automatically compute are. Thus, we propose the new keyword *concrete* to indicate that a field is computed in the concrete layer and two keywords to indicate its starting and maximal value. The maximal value is used for in fact resetting the value to the starting value when the maximum is reached.

```
type record MyPeriodicMessageType {
  integer myConcreteField concrete startvalue 0
                          maximumvalue 32000,
  // more fields definitions }
```

Here again, methods to access the start and maximum values should be provided by the *triAdapter* class in order for the scheduler to access them.

Updating Individual Field Values

Real time constraints of systems that use periodic messages call for minimizing the amount of data being manipulated especially in the codec. In particular, it is important

to avoid having to encode an entire record when only the value of a minimal amount of fields has changed. Thus, there needs to be a separate keyword for sending only a new value for a given field when all the remaining fields remain constant. We propose the keyword update with three parameters, one that indicates the data type, one for the field name and finally one for the new value as for example:

```
myPort.update(MyPediodicDataType, aFieldName, 45000);
```

This abstract layer keyword requires a corresponding method *triUpdate*() in the TriAdapter abstract class defined in the TTCN-3 standard [13] that must be implemented exactly like the *triSend*() method that corresponds to the abstract send() command. However, the update command has another challenge, it requires late coding since it affects only a small portion of the persistent data buffer used by the scheduler. Effectively the traditional model of sequence of actions –abstract send-message encoding-concrete *triSend* cannot be used since only a portion of a message is encoded with the updated value. Thus, the *triUpdate*() method will receive an abstract value in its parameter. The implementation of this method consists in making the appropriate modification to the data buffer corresponding to the specified data type and this remains the responsibility of the user.

Timeout and Out of Sequence Events

For receiving periodic message, we propose a concept of timeout event associated with the data type rather than a concept of timer at the abstract level. This is due partly to the fact that periodicity is an implicit timer that thus does not need to be specified as such but also because periodicity is by definition associated with a specific data type. Thus we propose an abstract language construct with the keyword *periodictimeout* that has a data type as an argument and that is associated with a port instance as follows:

```
myPort.periodictimeout(MyMessageType);
```

However, this timeout is triggered by a timer in the concrete layer that should be built-in and thus implemented by the TTCN-3 execution tool provider specifically for periodic messages only and not by the user.

Consequently, when a timeout occurs in the concrete layer, a timeout event should be en-queued on the message queue.

The setting or checking of out of order sequence numbers can be specified in a similar way. For checking out of sequence messages being received we use an out of sequence indicator event that indicates the data type and the field name in which sequence numbers are stored as follows:

```
myPort.outofsequence(MyMessageType, myFieldName);
```

However, when sending messages, while the periodicity is naturally controlled with the *setperiodicity* construct, provoking out of sequence messages for testing the response to such a situation by the SUT is not as simple. Here we need to specify

which sequence numbers need to be interchanged without really knowing at which state the sequence numbers are in since they are computed by the scheduler. Thus we propose a concept of relative position of sequence numbers which really indicates that at the present point, a sequence number that normally would occur only later should be sent immediately. We propose the following syntax for specifying out of order sequence numbers in terms of a relative distance between sequence numbers:

```
myPort.setoutofsequence(MyMessageType, myDistance);
```

For example, a relative distance of 8 would create the following suite of sequence numbers when they have already internally reached the value of 235:

{235, 243, 236, 237, 238, 239, 240, 241, 242, 244, …}

However, one important aspect of periodic messages is that some field values can be determined only at the concrete layer. This requires a mechanism to specify the encoding or decoding rule of such fields well decoupled from the value assignment mechanism. The user would specify the encoding/decoding rule while the value would be determined by a scheduler. This problem is novel and is future work.

4 Test Certification Using the TTCN-3 Matching Mechanism

Periodic message systems used in the aviation industry require certification by law. For example, no airplane can fly without a certification of navigability which includes certification of each of its components, whether hardware or software including test suites. There are numerous approaches to test certification used by industry. The decision factors for choosing the appropriate approach for testing are described qualitatively in a survey in [3]. They first distinguish the test system for its capacity for abstraction from a code-centric to a model-driven view and second discuss the advantages of test specification languages. TTCN-3 meets these criteria and has the additional advantage of being a standard; being well supported by a number of tool vendors; and by having a clear model. This raises the question of migrating from legacy test systems to TTCN-3. However, in light of the certification requirement, any migration from a legacy test system to TTCN-3 also requires certification of the corresponding new TTCN-3 test suite. Certification is a long and tedious process. Thus, migrating to a new test specification language requires reducing the certification process. Normally, this can be achieved by proving behavior equivalence.

The major advantage of using TTCN-3 as a formal test specification language is the separation of concerns between an abstract level in which test logic is specified and a concrete level in which system specific messages and protocols are parsed. In the case of test certification, this means that the equivalence can be determined at the abstract level only and using the very powerful and simple matching mechanisms that saves the test engineer the actual implementation of message comparisons. This task is handled by the built-in mechanisms in the TTCN-3 execution tool.

Behavior equivalence has been already researched for the case of refactoring TTCN-3 test suites. Makedonski et al [7] studied the equivalence of observable behavior of two TTCN-3 test suites, the latter being the result of refactoring the former.

They define observable behavior by the interactions of the test system running the test case and the SUT and the test verdicts and they came to the conclusion that the test system interface is the best location to observe behavior. More precisely, they state that the possible sequences of observable events define the observable behavior. Basically, behavior equivalence means the same message sequences, the same message content, and the same test verdicts. Practically, they use TTCN-3 logs to evaluate behavior equivalence. TTCN-3 logs have the advantage of providing message content in their abstract form which makes the use of the TTCN-3 matching mechanism possible with a valuable consequence of being able to evaluate the equivalence strictly in abstract terms since the codecs would remain constant by definition. Their study was based on discrete messages systems. In our case we face two new challenges:

- The comparison between a non-TTCN-3 and a TTCN-3 test system.
- Periodic rather than discrete messages.

The first difference makes the use of logs difficult because both the format and the content of these logs are heterogeneous and would require at least a translation prior to any attempt to use them. Thus, as a result, the only solution would be to capture concrete messages on the transport medium instead of abstract messages in logs. With periodic messages, the main problem is precisely the periodicity that produces large quantity of messages that need to be compared to prove equivalence. Here the principle of decoding messages only if their content has changed as shown previously on Fig. 5 can be used efficiently for the purpose of certification because it reduces the number of messages to be compared enormously and also avoids interleaving of different message types. Thus, with this approach we focus on the changes in state of the messages. This results in a more manageable reduced number of messages that is then comparable to what can be found in discrete messages systems.

The approach consists of using the messages from the legacy system as test oracles against the messages produced by the corresponding TTCN-3 implementation. The verification of equivalence can be performed by another TTCN-3 test suite that uses the logs of both the legacy and TTCN-3 migration test suites and re-uses the codecs of the migrated TTCN-3 test suite as shown on Fig. 7. Thus, the TTCN-3 certification test suite is composed of sequences of receive statements only since it checks both captured messages being sent or received either by the legacy or migration test suites as shown on Fig. 8.

Thus, the certification process is performed in five steps using TTCN-3:

- Capture concrete messages produced by the execution of the legacy system using a packet sniffer such as Wireshark [21].
- Transform the concrete legacy messages into a list of abstract TTCN-3 messages templates using the codec.
- Capture concrete messages produced by the execution of the TTCN-3 migrated system using a packet sniffer again.
- Transform concrete messages produced by the TTCN-3 migration test suite into a list of abstract TTCN-3 messages templates using the same codec as in step 2.

- Use a simple comparator implemented in the abstract layer of TTCN-3 that compares an abstract message of the legacy system to its corresponding abstract message of the migrated TTCN-3 system using the basic TTCN-3 matching mechanism.

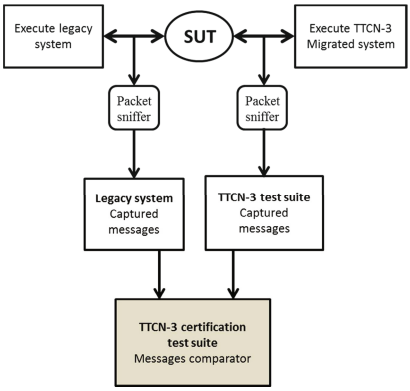

Fig. 7. Using TTCN-3 for certification

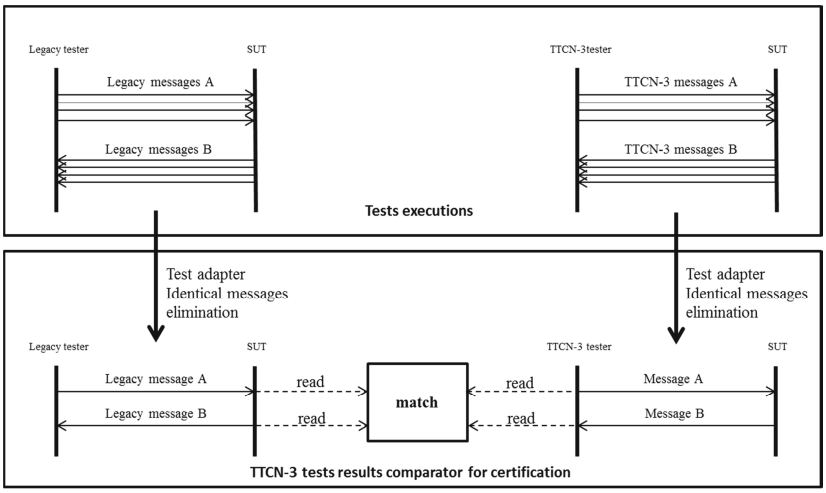

Fig. 8. Test Certification message comparison

Since both test systems use different message types, the TTCN-3 messages comparator test behavior is composed of alternatives (line 04) for each data type on the port handling the legacy messages. A message received on the legacy port (line 05) is stored into a variable (line 06) that is used as a template (line 09) when attempting to receive and match the corresponding message from the port handling the migrated messages as shown below:

```
01 function messageComparator() runs on ComparatorType{
02  var template MessageType_A legacyMessage_A;
03  var template MessageType_B legacyMessage_B;
04  alt { // code handling MessageType_A
05     [] legacyPort.receive(MessageType_A:?)
06                              -> value legacyMessage_A {
07       legacyMessage_A.sequenceNumber := ?;
08       alt {
09          [] migrationPort.receive(legacyMessage_A){
10              setverdict(pass);
11          }
12          [] migrationPort.receive {
13              setverdict(fail);
14              stop;
15       }   }
16       repeat;
17     }
18     // other alternatives for different message types
19 }  }
```

Note that the variable must be a template so that the sequence number of that message can be set to any value using the "**?**" symbol (line 07) to avoid matching this field since sequence numbers are not handled in the abstract layer for periodic messages. Thus, here we can see that the TTCN-3 language and its central model of separation of concern are used in different ways in the process of certification because this model ensures a maximum flexibility and the high level of abstraction combined with the built-in matching mechanism of TTCN-3 tools makes the specification of this message comparator extremely simple.

5 Verifying Requirements Using TTCN-3

5.1 Defining Verification of Requirements

While most test systems are test case centric, the aviation industry uses an additional level of structuring, the concept of requirements that consists in enumerating selected test cases that should have passed to satisfy a given requirement. Such requirements are described in standards such as in the joint FAA/EASA DO-178C standard [12] and related documents which also cover the quality assurance process (testing) [10]. They target, in particular, model based development and verification using formal methods. The requirements are based on a metric of severity that consists of five different levels ranging from no effect to catastrophic. This means that a system may still be authorized to be put into production even if a requirement with a low severity level is not full-filled, i.e. some test cases composing it have failed. For example, an aircraft may still be authorized to fly if the test for the functioning of the entertainment system is not full-filled. The evaluation of the fulfillment of requirements is performed in two

steps: first, all the test cases of a test suite are executed independently from their involvement in a requirement, second, each requirement is evaluated using the subsets of test cases that define them. This means that a given test case could belong to several requirements and thus is evaluated but not executed several times. The failure of a test case could thus have different impacts depending of the severity associated with the requirements it belongs to.

5.2 Implementation Details

The verification of requirements can be easily implemented in TTCN-3 using the verdicts that are returned by test cases that are executed in the control part. First, requirements are defined using sets of records containing two fields, one for the requirement name and the other for the set of test cases containing the test cases names and their corresponding desired verdicts (always pass) as follows:

```
type record TestcaseVerdictType {
  charstring testcaseName,
  verdicttype testcaseVerdict
}
type set of TestcaseVerdictType TestCasesVerdictsType;

type record RequirementType {
    charstring name,
    TestCasesVerdictsType testCasesVerdicts
}
type record of RequirementType RequirementsType;
```

Using the above abstract data types, we can define the requirements using TTCN-3 templates as follows:

```
template RequirementsType requirements := {
  { name:= "requirement_1",
    testCasesVerdicts := {
     {testcaseName:= "TC_1", testcaseVerdict := pass },
     {testcaseName:= "TC_2", testcaseVerdict := pass },
     {testcaseName:= "TC_3", testcaseVerdict := pass }
     }
  },
  { name:= "requirement_2",
    testCasesVerdicts := {
     {testcaseName:= "TC_1", testcaseVerdict := pass },
     {testcaseName:= "TC_3", testcaseVerdict := pass }
} } }
```

Then, we need to execute the test cases independently from the definitions of requirements and collect their returned verdicts in variables of type *verdicttype*. The test case verdicts variables must be initialized to the standard value *none* in order to avoid

problems resulting from uninitialized variables in the case of conditionally executed test cases where the actual execution of the test case is not guaranteed. The control section is executed as follows:

```
control {
  var verdicttype verdictTC_1 := none;
  var verdicttype verdictTC_2 := none;
  var verdicttype verdictTC_3 := none;
  verdictTC_1 := execute(TC_1());
  verdictTC_2 := execute(TC_2());
  verdictTC_3 := execute(TC_3());
```

Once all test cases have been executed, we compose a template variable to create a set of TestCaseVerdictType that is defined as a subset and contain the actual verdicts resulting from executing the test cases individually:

```
var template TestCasesVerdictsType tc_verdicts :=
   subset (
     { testcaseName := "TC_1",
       testcaseVerdict := verdictTC_1 },
     { testcaseName := "TC_2",
       testcaseVerdict := verdictTC_2 },
     { testcaseName := "TC_3",
       testcaseVerdict := verdictTC_3 }
   );
```

Finally we can verify the requirements by using the TTCN-3 matching mechanism construct (keyword *match*) for sets using the requirements and test cases verdicts templates that were defined previously, one statically and the later dynamically as follows:

```
var integer i;
for(i:= 0; i < sizeof(requirements); i := i + 1) {
  if(match(requirements[i].testCasesVerdicts,
                                    tc_verdicts)) {
    log("requirement " & requirements[i].name &
                          " has been fullfilled");
  }
  else {
    log("requirement " & requirements[i].name &
                         " has NOT been fullfilled");
} }
```

Thus, the principle consists in verifying that the set of test cases verdicts of a requirement is a subset of the entire set of test cases verdicts. The match can occur only if all of the test cases of a given requirement have passed. As a result, test cases that have failed do not influence the evaluation of a requirement if they are not specified for this specific requirement.

5.3 Recommended Change to the TTCN-3 Standard

While the implementation described in the previous section certainly functions correctly and is straightforward, it requires a development effort and this not only on the requirement evaluation activity but also on the test report aspect. Test reports are not covered by the standard and are effectively provided with various levels of details by TTCN-3 tools. Thus, here we can only recommend the specification of requirements at the abstract level leaving responsibility of implementation of the corresponding logging to the tools providers. The abstract language construct can be relatively simple: the keyword *requirement* followed by a requirement identifier and a list of test cases identifiers as follows:

```
requirement myRequirement
                        {myTestcase_1,...,myTestcase_n};
```

The execution of test cases would still have to be specified separately within the control section and could include conditional execution. Although this proposal would be adequate for the industrial avionics systems we have studied because their test cases are not parametric and thus a given test case is executed only once in a test campaign, TTCN-3's parametric test case concept allows the execution of the same test case but with different parameter values. Thus, the TTCN-3 test case name as a way to identify an effective test case is not sufficient. Solutions to this problem are future work.

6 Conclusions

In this paper, we have shown how TTCN-3 can be adapted to provide the same level of efficacy in testing periodic messages in avionics systems as is provided for discrete messages in telecommunications systems. The key insight is to leverage the separation of concerns in the architecture of TTCN-3 between the abstract test specification layer that specifies test behavior and the concrete messaging layer that filters and interprets the periodic messages. This approach can be implemented with the current version of TTCN-3, but we have suggested some simple extensions to the TTCN-3 standard that would support testing of periodic message protocols in a more natural fashion. Test certification and verification of requirements are critical aspects of avionics systems governed by standards. We have shown how our approach to periodic messages is applicable to them as well.

References

1. ETSI ES 201 873-1 (2013-04). The Testing and Test Control Notation version 3, Part 1: TTCN-3 Core notation, V4.5.1 (April 2013)
2. Testing Technologies, TTworkbench - an Eclipse based TTCN-3 IDE (2011), http://www.testingtech.com/products/ttworkbench.php
3. Hartman, A., Katara, M., Olvovsky, S.: Choosing a Test Modeling Language: A Survey. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 204–218. Springer, Heidelberg (2007)

4. Schieferdecker, I., Bringmann, E., Großmann, G.: Continuous TTCN-3: Testing of Embedded Control Systems. In: Proceeding of SEAS 2006, pp. 29–36 (2006)
5. Efkemann, C., Peleska, J.: Model-Based Testing for the second generation of Integrated Modular Avionics. In: Proceedings of ICSTW 2011, pp. 55–62 (2011)
6. Laurent, O.: Using Formal Methods and Testability Concepts in the Avionics Systems Validation and Verification. In: proceedings of ICST 2010, pp. 1–10 (2010)
7. Makedonski, P., Grabowski, J., Neukirchen, H.: Validating the Behavioral Equivalence of TTCN-3 Test Cases. In: Proceeding of VALID 2009, pp. 117–122 (2009)
8. Audsley, N.C., Grigg, A.: Timing Analysis of the ARINC 629 Databus for real-time applications. Microprocessors and Microsystems 21(1-7), 55–61 (1997)
9. Gabillon, A., Gallon, L.: Availability of ARINC 629 Avionic Data Bus. Journal of Networks, Vol 1(6), 1–9 (2006)
10. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. IEEE Software (May/June 2013) (issue)
11. Easwaran, A., Lee, I., Sokolsky, O., Vestal, S.: A Compositional Scheduling Framework for Digital Avionics. In: University of Pennsylvania Scholarly Commons, Departmental Paper (August 24, 2009)
12. DOC-178C, Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc., `http://www.rtca.org/store_product.asp?prodid=803` (last accessed August 2013)
13. ETSI ES 201 873-5 (2013-04). The Testing and Test Control Notation version 3, Part 5: TTCN-3 Runtime Interface, V 4.5.1 (April 2013)
14. ETSI ES 202 781 TTCN-3: Extension: Configuration and Deployment Support, V 1.2.1 (June 2013)
15. ETSI ES 202 782 TTCN-3: Extension: Performance and Real-Time Testing, V 1.1.1 (July 2010)
16. ETSI ES 202 783 TTCN-3: Extension: Testing of Real-Time Systems, V 1.1.1, draft
17. ETSI ES 202 784 TTCN-3: Extension: Advanced parameterization, V 1.3.1 (April 2013)
18. ETSI ES 202 785 TTCN-3: Extension:Behavior Types, V 1.3.1 (April 2013)
19. ETSI ES 202 786 TTCN-3: Extension:Support of Interfaces with Continous Signals, V 1.1.1 (April 2012)
20. ETSI ES 202 789 TTCN-3: Extension: Extended TRI, V 1.2.1 (April 2013)
21. Wireshark software, `http://www.wireshark.org/`
22. Levine, D., Gill, C.D., Schmidt, D.C.: Dynamic Scheduling Strategies for Avionics Mission Computing. In: Proceedings of Digital Avionics Systems Conference, vol. 1, pp. C141–C158 (1998)
23. Hua, Y., Liu, X.: Scheduling Design and Analysis for End-to-End Heterogeneous Flows in an Avionics Network. In: University of Nebraska, Digital Commons, CSE Conference and Workshop Papers (2011)