periodica polytechnica

Electrical Engineering
51/1-2 (2007) 11–19
doi: 10.3311/pp.ee.2007-1-2.02
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

A test suite conversion method: from interoperability test to conformance test

Sarolta Dibuz / *Péter* Krémer

Received 2005-03-31

Abstract

In this paper we propose a method for easier creation of conformance test suites. Instead of a formal description of a protocol, our method needs an interoperability test suite. The conformance test suite in this method is constructed by re-using parts of an existing test suite. We have used a test suite for OSPF (Open Shortest Path First) routing protocol to present our method through an example. We describe how the interoperability test suite and the conformance test suite for the same protocol look like. Then we construct the re-using method by comparing these test suites and by identifying the re-usable parts. We also investigate the resulted test suite to find out the goodness of the conversion. Finally, we compare the coverage of two conformance test suites that are based on the same test purposes and one of them was created by this conversion method.

Keywords

interoperability test · conformance test · TTCN-3 · test suite conversion

Sarolta Dibuz

Ericsson Telecommunications Hungary, P.O. Box 107, H-1300, Budapest 3, Hungary

e-mail: Sarolta.Dibuz@ericsson.com

Péter Krémer

Ericsson Telecommunications Hungary, P.O. Box 107, H-1300, Budapest 3, Hungary

e-mail: Peter.Kremer@ericsson.com

1 Introduction

Interoperability testing checks if two different implementations of the same protocol have the capability of inter-working. It is used for testing prototypes built on RFCs and products implementing Internet protocol standards. Interoperability testing is a popular verification method in IETF (Internet Engineering Task Force). A protocol draft can be an IETF standard only if there exists at least two inter-operating implementations for it. That is, interoperability testing is such a well accepted method by standardization bodies that they use it to check the correctness of a specification. Unfortunately, there can be common errors in implementations or misunderstandings of the protocol description. If two IUTs (Implementation Under Test) have the same error of this kind, this error will not be discovered by interoperability testing.

Conformance testing is used to check that the implementations conform to the protocol specification [1]. In the telecom world conformance testing is more applied. ETSI, ITU, 3GPP and other standardization bodies develop conformance test suites. Vendors of telecom equipments or operators – the customers – are used to apply these conformance test suites to show conformance of the products or for type approval. Interoperability test is also done after the conformance tests. Its main function is to check if a new network element can inter-operate with the other nodes of the network on the main operation level. Conformance test suites cannot cover 100% of the protocol's functionalities. It may happen that interoperability test of two IUTs fails even if the IUTs passed the conformance test. That is why interoperability testing is also needed beside conformance testing. As it can be seen, the two kinds of test suites are for different purposes and both of them are needed to thoroughly test a protocol implementation.

So, if we have to test a protocol, we need an interoperability test suite (ITS) and a conformance test suite (CTS), too. But it is not efficient to write two separate test suites for the same protocol. The best solution would be to create a kind of formal description of the protocol and then generate both test suites from this description. Theoretically, it is a good solution but unfortunately not feasible in practice. There are only a few com-

mercial tools available that support this kind of generation (e.g. Telelogic Tau). Furthermore, only CTS generation is possible at the moment. Another problem is that the protocols are usually too big and these tools generate huge test suites. They are difficult to execute in practice and they contain many test cases, which are not relevant or important to execute. That is why it is quite rare in practice that testers build a formal model first and then use some methodology to generate the test cases automatically. However, the theory is undoubtedly beneficial, it is not widely used. Instead of using formal models, we have tried to find something else that is already available or easier to create for an average tester.

We have already shown in a previous paper [7] that an ITS is easier to derive from a textual description than a CTS. For example, the inputs and stimuli are not specified as protocol messages, the protocol specification is not needed to be so mature, etc. Moreover the development of an ITS can be an iterative process. I.e. the monitoring functions are developed later, when several log files are available and the message templates and parameters can be extracted. Our experience also shows that the need is much higher for an ITS during the development of a protocol implementation and testers usually write some kind of test suite to fulfil their own needs for testing.

Both ITS and CTS have to be written for testing protocol implementations. If we write an ITS and then we derive the CTS from it, we could save a lot of time and effort. Let's investigate whether they have some common or at least similar parts.

In this paper, we describe the way a protocol implementation is usually tested in Section 2. Section 3 presents the method we propose to follow in order to re-use an ITS to get the CTS. In the next Section we compare the two test suites and show the different and the identical parts. It is also shown how to re-use the appropriate parts of an ITS in a CTS. This process is described in Section 6. The resulted CTS is investigated in Section 7 to find out the goodness of the conversion. Then, we compare this test suite to another CTS to verify the coverage of test purposes in the converted one. Finally, we show a small example to present this method in practice.

2 The current practice

In this Section we present the current practice that is applied for protocol testing. According to it, the typical testing activities of a protocol implementation are the following:

1 ad-hoc interoperability test (during implementation development)

2 specification of test purposes

3 CTS development

4 conformance testing

5 ITS development

6 interoperability testing

At first sight, it seems that every step in this process is needed to make both the developer (or tester) of the implementation and also the customer satisfied. The developers need some method to check the implementation during the development of the implementation. Since an ITS is much easier to write [7], the interoperability test (as noted in the first step) is quite common. The customer needs some kind of evidence that the implementation is correct and behaves just like it is described in the protocol specification. That is why the second, third and fourth steps are needed. There are also cases when the customer also require an interoperability test to see if the implementation not only conform to the specification but can inter-work with other protocol implementations, as well. In this case the fifth and the sixth steps are also inevitable.

But by taking a deeper look at this approach, we can see some problems, especially when we take into account the efficiency. For example, the first interoperability test is performed by the developers and the second one is written and executed by the testers. They have different knowledge, they use different methods and tools: developers write some testing functions in the source code, while testers usually do not alter the implementation to test is.

Our aim with this work was to reduce required man-power for all of these efforts by identifying the common parts of an ITS and a CTS and by re-using some parts of the test suites (with or without modification) would also be a considerable benefit. Of course, the current practice also needs to be changed to facilitate the development of reusable test suites. The detailed method how to write such a test suite is described in the next Section.

3 Method for re-using test suites

As it can be seen in Section 2, the current practice leaves some space for improvements. At first, in order to make it possible to easily re-use test suites we need a common description language. Since we do not want to convert between different languages and would like to re-use parts of test suites as easy as possible, we need a common language in which we write our tests.

Nowadays, TTCN is a common language for writing CTSs and its latest version [2] is general and flexible enough to describe ITSs, as well. If the two different types of test suites are written in the same language then some parts of it can be simply copied. This situation becomes more and more likely as TTCN–3 based test tools get used widespread. The interoperability test is part of the protocol design (at least in IETF), protocol specification, implementations and tests are developed together. It also means that an ITS is written before the CTS and both can use the same description language. That is, results, experiences and parts of the TTCN–3 source code of the interoperability tests can be re-used in conformance tests. These parts are the definitions of the protocol's data types and messages, timers, templates, altsteps[1] and functions that describe the dynamic be-

[1] An altstep is a method to choose among several alternative events.

haviour of a protocol. Because of these reasons and because TTCN–3 has already proved its usability in several areas of testing, we have chosen TTCN–3 as the common language.

The following steps describe the method we propose:

1 specification of test purposes

2 ITS development

3 interoperability testing

4 CTS development by re-using parts of the ITS

5 conformance testing

6 interoperability testing

In the first step, the testers write the specification of the test purposes. This step is already part of the current practice (so it does not require any extra work) we just moved it into the beginning. The reason is simple: both test suites should be based on the same test purposes in order to get a similar structure, which makes the re-use more simple. Since both test suites are written for the same protocol, it is obvious that the same test purposes can be used. In the resulted test suites one test case is written for one test purpose. However, it is also possible to use different structure for the test suites, e.g. write an ITS where one test case checks more than one test purposes. Reusing is also possible in this case because the information can still be extracted from the interoperability test case (see Section 7 for more details). The only difference is that this information will be used to construct more than one conformance test cases and not for only one. Either way the ITS is written it is correct and fully functional but if the ITS and the CTS are based on the same test purposes, it is much easier to re-use some parts.

The second step uses the same efforts as needed for the ad-hoc interoperability tests. This ITS is a result of an iterative development just like the old one, it grows as the implementation is getting better and better. The difference here is that the ITS is already written according to the test purposes. The result in this step is an ITS that is developed parallel with the implementation, so it can be used for testing the implementation during development. Even the first working version can be tested, as the test suite is already available at such an early phase. Moreover, the very same ITS can be used for the interoperability test required by the customers in step 6.

To reach this point we only used basically the same effort that is needed for ad-hoc interoperability test and for test purpose specification, according to Section 2. Thus, we already saved the work needed to develop a separate ITS. The *only* thing we still need is a CTS. Now, instead of writing the CTS from scratch, we construct it by simply copying parts of the ITS. What we get at the end is a CTS, which is exactly the same as if it was developed from the test purposes. Of course, there can be such test purposes that cannot be implemented in an ITS. In this case the test cases for the missing test purposes must be written somehow. Do not forget though, that all the templates and supporting functions for the rest of the test suite are already available. Usually, several test cases call the same functions and use the same templates, thus the availability of these items makes this task quite simple.

In the next subsections we present two test suites: one ITS and one CTS. First, we have written the test purposes, then the ITS, and finally the CTS. That is, the two test suites are based on the same test purposes.

3.1 Interoperability Test

For interoperability test we have used the MAIT (Model for Automated Interoperability Test) model [7], thus in the following pages we will refer to a MAIT test suite when we refer to ITS. A MAIT test suite can be represented in TTCN–3, implementation-independent, suitable for automated tests and applicable for any kind of protocol.

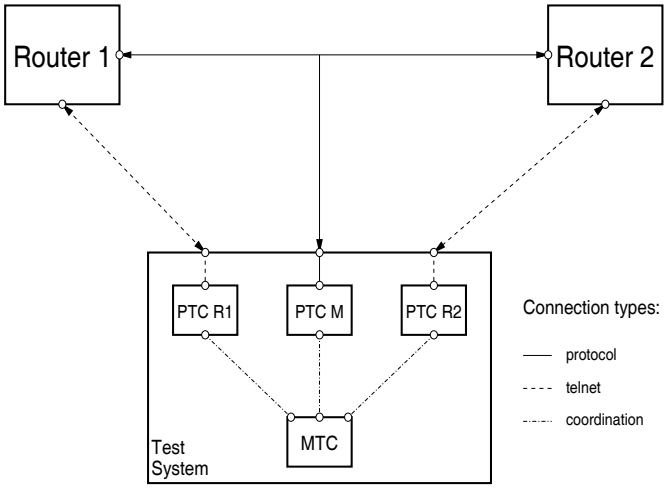

Fig. 1. Test configuration for interoperability test of OSPF Hello protocol

Fig. 1 shows the test configuration for interoperability tests. The configuration consists of *Router 1*, *Router 2* and the *Test System*. *Router 1* and *Router 2* denote the two OSPF implementations, which are under test. The *Test System* handles the following tasks:

- remote controlling of *Router 1* (*PTC R1*),
- remote controlling of *Router 2* (*PTC R2*),
- monitoring the network (*PTC M*),

Since these tasks are independent from each other, they are implemented in Parallel Test Components (PTC). The Main Test Component (*MTC*) is used to coordinate the behaviour of PTCs.

PTC R1 establishes a telnet connection between *Router 1* and the *Test System*, this connection is then used to remote control *Router 1*. The test component emulates an ordinary user: configures the IUT, starts an application or triggers a special test by giving the appropriate input. These kinds of inputs cannot be given through the IUT's standardized interfaces because most of the protocol standards do not specify the upper interface. Basically, it configures, starts and makes everything what only an experienced developer of that particular implementation can do.

PTC R2 creates the same type of connection but its tasks are slightly different. It also has to start and configure the implementation but then it checks if the test ran correctly and the necessary changes were made. It emulates the user of the other implementation, who follows the tests at the console and sees if everything is working correctly or not. It can be the establishment of a connection, a new record in a database or something else which shows that the test was successful. The messages that are sent on this type of connection highly depend on the implementations. In order to avoid the re-compilation of the test suite, if a new implementation is tested, these messages are given as test suite parameters.

All testers like to know what happens on the wire, thus they always monitor the network that connects them to the other implementation. In general, "tcpdump" (or a similar tool, e.g. ethereal) is used to record the packets. Synchronization of starting and stopping the packet recorder tool and the analysis of the log files are made manually in most of the cases. We have defined a separate test component in our model to handle and to automate these tasks. *PTC M* monitors the network, records and analyses every protocol message that the implementations send.

The function of *MTC* is to synchronize and to co-ordinate the behaviour of PTCs. Similarly to conformance testing, the execution of a test case is divided into 3 phases (the names of the involved PTCs are shown in parentheses):

1 Configure the implementations for the test case (*PTC R1*, *PTC R2*).

2 Execute the test case (*PTC R1*, *PTC R2*, *PTC M*).

3 Restore the original configuration (*PTC R1*, *PTC R2*).

These phases are separated in such a way that each one is implemented by a function call. Thus, a MAIT test suite consists of two main parts, one part is a MAIT skeleton, that is common for every MAIT test suite and creates the PTCs, handles the communication between MTC and PTCs and so on. This part is referred as MAIT-static because it is the same in all MAIT test suites and this part is protocol-independent. The second part contains the functions that configure the implementation, execute the test cases and restore the original configuration. This part is called MAIT-dynamic because the behaviour to check the IUTs is described here. In the remaining pages of this paper we take into account MAIT-dynamic only because this is the part that is related to the protocol.

We assume that each test case in an ITS checks only one test purpose. This assumption makes the re-use of our ITS more user-friendly. Re-using is also possible if this condition is not fulfilled, the details can be found in Section 7.

3.2 Conformance test

The test configuration for the conformance test of OSPF's Hello protocol (Fig. 2) is more simple than the interoperability test. In this case, there is only one test component needed: the *MTC*. It has two ports, one for OSPF messages and the other one is a telnet connection, which is used to transmit configuration commands and messages for the upper tester (e.g. start/stop the implementation).

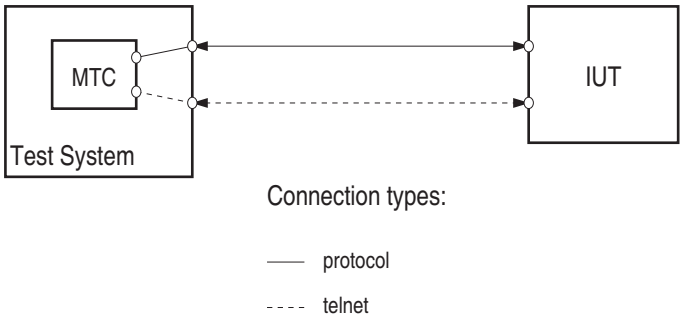

Fig. 2. Test configuration for conformance test of OSPF Hello protocol

In a typical conformance test case, we have to do the following tasks:

- configuring the IUT,
- checking the test purpose
- driving the IUT back to its original state

These tasks must be performed one after another, so they do not need more PTCs. Depending on the complexity of the protocol, there may be other PTCs existing during the check of a test purpose. The Hello protocol is not so complicated that it would require more PTCs than the *MTC*. So, our *Test System* consists of one test component (*MTC*) and two ports. The *MTC* has the very same ports, thus it will have the same component type as the *Test System*. In the CTS each test case checks exactly one test purpose.

4 Comparison of two test suites

In this section we will compare our test suites and will identify the identical or similar parts. First, we describe the notation that we will use in the following pages. Let TS denote a TTCN–3 test suite that is based on test purpose TP as a 5-tuple:

$TS(TP) = (D, C, TI, T, B)$, where

D : Definitions

C : Constants

TI : Timers

T : Templates

B : Dynamic behaviour descriptors

D denotes all types of definition, in TTCN–3 they can be data, component and port type definitions. Let D^{data} denote data types, D^{comp} component types and D^{port} port types. An additional CTS or ITS index will denote a CTS and an ITS. For example D^{data}_{CTS} stands for the data types in a CTS. Since

the two test suites are for the same protocol, it is pretty obvious that the data type definitions must not differ. Thus we can say that $D^{data}_{CTS} = D^{data}_{ITS}$. If these definitions are stored in a separate TTCN–3 module, then the same module can be imported in both test suites. The test suites also use the same interfaces to interact with the IUT(s) and such interfaces are represented by ports in TTCN–3. That is, the port type definitions must also be the same in both test suites: $D^{port}_{CTS} = D^{port}_{ITS}$. As it can be seen on Fig. 2, the components are quite different: $D^{comp}_{CTS} \neq D^{comp}_{ITS}$.

The C represents the global constants[2] and the parameters (that are constant during the execution) of a test suite. They have a common attribute: all of them must have a specific value, so wildcards and matching mechanisms cannot be used. Constants are used to store values defined in the protocol specification that does not change at all, e.g. the version of the protocol. Another typical usage of constants is the case when certain specific values are stored in order to improve the readability of the test suite. For example, there is a special multicast address (e.g. AllSPFRouters) where the initial Hello packet needs to be sent. Regardless the purpose of the constants, they are all protocol-related, thus their value does not depend on whether they are used in a CTS or in an ITS. Denoting the constants with C^{const}_{CTS} for a CTS and with C^{const}_{ITS} for an ITS: $C^{const}_{CTS} = C^{const}_{ITS}$. Parameters ($C^{par}$) describing the behaviour of an implementation in interoperability test must also be given for a CTS. It is because the same kind of information is needed to properly set up a conformance tester. So, we can say that the parameters of a CTS and an ITS are the same: $C^{par}_{CTS} = C^{par}_{ITS}$, thus $C_{CTS} = C_{ITS}$.

The timers (denoted by TI) used in a test suite are heavily determined by the protocol. Since the protocol does not change, the timers should not change as well: $TI_{CTS} = TI_{ITS}$.

Templates (T) describe the abstract data that we would like to send and receive in a TTCN–3 test suite. In this test suite we use two types of messages: protocol messages (OSPF packets) and configuration messages (over a telnet connection). The configuration messages are exactly the same in both cases, so the corresponding templates should be the same too. But there is a small difference between the templates of protocol messages in this case. During interoperability test, we do not send any protocol message, so there are only receiving templates in the ITS. That is, $T_{ITS} \subset T_{CTS}$ and $T_{ITS} = T_{CTS} \setminus T^{ps}_{CTS}$, where T^{ps}_{CTS} denotes templates that are used to send protocol messages. However, the latter set (T^{ps}_{CTS}) may be empty if the receiving templates can be used for sending, as well.

Dynamic behaviour descriptors (B) include all the constructs that are used to describe the protocol behaviour. They are the altsteps and defaults (B^{alt}), functions (B^{func}) and test cases (B^{tc}). Altsteps and defaults are handled together because they are defined in the same way in TTCN–3. The only difference between them is the method they are used in the test suite they are both a shorthand to group multiple alternatives together. Altsteps behave like functions but for receiving events, one can process several kinds of incoming packets by defining an altstep. A message sequence that is handled by the ITS must also be handled by the CTS. The message sequences determined by the protocol specification are already described by altsteps in *PTC M*. Since a CTS must also work based on the protocol specification, the altsteps of an ITS can also be used in a CTS, so $B^{alt}_{CTS} \subset B^{alt}_{ITS}$.

One part of the functions is the same in the two test suites (e.g. for configuring the IUTs), $B^{func}_{CTS} \subset B^{func}_{ITS}$. The other part of the functions in an ITS describe the dynamic behaviour of a protocol, which is normally part of a test case in a CTS. As a consequence of the different architecture and purpose of the two test suites, the test cases are completely different. But the test cases of a CTS include the information needed for the corresponding functions of the ITS: $B^{tc}_{CTS} \cap B^{func}_{ITS} \neq \emptyset$, so re-using is still possible. Moreover, the test cases in the MAIT model belong to MAIT-static, which does not change test suite by test suite, so the interoperability test cases can be re-used in another ITS.

5 Conversion of test suite parts

This section describes how and why it is possible to convert parts of an ITS to a CTS. Most of the parts of a test suite are reusable without modification (D, C, TI) or easy to convert (T). Only the conversion of elements in dynamic behaviour descriptors (B) may need further explanation. Namely, the details of $B^{tc}_{CTS} \rightarrow B^{func}_{ITS}$ are shown in this section.

Suppose we have a test purpose tp_i. If it is possible to check that test purpose with an interoperability test case, then we already have the functions $B^{func}_{ITS}(tp_i)$ that describe the protocol's behaviour in that case. It is obvious that there will be a test case in the CTS that checks the same test purpose, denote it with $B^{tc}_{CTS}(tp_i)$. We know that $B^{func}_{ITS}(tp_i)$ must provide the following functionalities:

1 configuring the IUT

2 stimulate the IUTs

3 checking the message sequences between the IUTs

4 checking that the test was successful

5 restoring the IUTs to their original states.

The first and the last steps are exactly the same in a conformance test case, as well. The stimuli in the second step are generated by the IUT(s) in case of interoperability test, which can be recorded to help the creation of the corresponding template and the reconstruction of the stimuli in the corresponding conformance test case $B^{tc}_{CTS}(tp_i)$. The message sequence described in the third step is checked by the function that runs on *PTC M*. Knowing the test purpose and the PTC where the function runs

[2] Global constants are visible in the whole TTCN–3 module. There are also local constants, which are visible only inside the statement block (e.g. component, function, test case) where they are defined. Local constants are considered to be part of that statement block. Since there are only local variables, they also considered to be part of their own statement block.

Tab. 1. Size of test suite elements in the OSPF test suites

Element type	common	Interoperability test suite	Conformance test suite
D^{data}	255 (39)	0 (0)	0 (0)
D^{port}	4 (2)	0 (0)	0 (0)
D^{comp}	0 (0)	12 (3)	3 (1)
C^{const}	1 (1)	0 (0)	0 (0)
C^{par}	0 (0)	2 (2)	2 (2)
TI	1 (1)	0 (0)	1 (1)
T	83 (4)	0 (0)	275 (6)
B^{alt}	10 (1)	20 (2)	0 (0)
B^{func}	182 (7)	1270 (24)	0 (0)
B^{tc}	0 (0)	0 (16)	875 (17)

we can unambiguously identify the function $B_{ITS}^{func,PTC\,M}(tp_i)$ that we need. Most of the information we need to construct $B_{CTS}^{tc}(tp_i)$ can be found in $B_{ITS}^{func,PTC\,M}(tp_i)$. All the messages that are exchanged between two IUTs are described in this function, so we will know what messages to send and what will be the answer. Not only the message types but the contents of the messages can also be extracted from the log files of the interoperability test. The same environment, the lack of conversion from a textual representation to another one that is used in the test tool also helps to reduce the possibility of misunderstandings.

6 Re-usability in practice

According to the method presented in Section 3, we show how to derive a CTS by using an ITS. First, we have written the test purposes for OSPF's Hello protocol and then developed the ITS. Then the CTS has been constructed by re-using the already written ITS.

Table 1[3] shows a brief summary of the results. The first column indicates the type of the elements of a test suite as it was defined in Section 4. The second column contains the size and the number of test suite elements that are exactly the same in the two test suites (the common parts). The third and fourth columns indicate the size (measured in source code lines) and the number of each additional element (written in parentheses) that are only present in an ITS or in a CTS. Thus, the number of elements in an ITS (CTS) is the sum of the values in second and third (fourth) columns.

As it can be seen, the data type definitions (D^{data}) of the ITS can be re-used in the CTS without any modification or conversion. There are 39 types defined in TTCN–3 for the OSPF protocol, which type definitions take 255 lines in the source code. As TTCN–3 is modularized (i.e. a test suite may consist of several modules and each module can be a part of any other test suite), the data type definitions can be stored in one module and can be imported to any test suite. The very same is true for the port type definitions (D^{port}).

The ITS uses 3 different types of components (D^{comp}): one for protocol messages, one for upper tester purposes (and for configuration) and the third one is for the Test System Interface. Unfortunately, the component type definitions are different in the CTS. But they only define the ports that the components are using and the port types are available, thus the new components can be constructed easily.

The ITS used only one constant (C^{const}), thus the same can be re-used in the CTS. We said in Section 4 that the parameters are identical. Although, Table 1 shows as if the test suites had different parameters, they have the same parameters with respect to the type and the number. We have only changed their names in order to adjust them to the CTS. The parameters have a particular type that describes the attributes of an implementation, for instance the IP address or the network mask. We call the parameters *IUT1*, *IUT2* in the ITS and *TESTER*, *IUT* in the CTS. That is, we consider the equation $C_{CTS}^{par} = C_{ITS}^{par}$ to be still true.

There is one timer that is common in both test suites and this is the only timer that the ITS uses. The other one has been added to the CTS later because it was needed in such test cases, which do not have counterpart in the ITS. It can happen if a test purpose is not applicable for interoperability test. The conformance test cases for such test purposes have to be created by some other method (see Section 7 for more details).

The number of Templates (T) heavily depends on the writer of the test suite. Their number can be extremely small if only one template is written for one message type (in this case every template will have a lot of parameters). The other extreme situation is when the templates have no parameters at all. Of course, the ideal situation is between them. In the ITS we have 2 templates for base protocol messages and another 2 that are used by the base templates. These templates are only applicable to receive packets, and these are used in the CTS. But we would also like to send packets, so we had to define 6 more templates. One of them is a base template that is used for sending a general message, each one of other four adds one more parameter to this template. The sixth one is a template for receiving purposes but it is used in a test case which is based on a test purpose that is not applicable for interoperability test. However it is inevitable to send protocol messages in a CTS, it is not necessary in an ITS. In case of interoperability test, we stimulate the IUTs by upper tester messages (in case of OSPF, these commands are sent over a telnet session). Then, the protocol messages are created and sent by the IUTs. In our MAIT (interoperability) test suite, these messages are monitored and logged by *PTC M*. Later, the log files can be used to construct the protocol messages and the templates that are needed for conformance testing. Though, the ITS does not provide the templates for sending protocol messages (denoted by T_{CTS}^{ps}) but makes it easier for the tester to create them.

[3] The code size of the interoperability test cases are not indicated because the test cases are considered to be in the static part of a MAIT model, and are not relevant in this Table. There is one test case missing in the ITS because it cannot be checked by interoperability test.

Altogether, there are three altsteps (B^{alt}) used in the ITS and one of them is re-used in the CTS, as well. Besides, there are 31 functions in the ITS, out of which 7 can be re-used in the CTS without any modification. We can divide these functions into two sets: functions that do not need any modification ($B_{ITS}^{func,NM}$) and the ones that may be altered ($B_{ITS}^{func,M}$). Naturally,

$B_{ITS}^{func,NM} \cup B_{ITS}^{func,M} = B_{ITS}^{func}$,
$B_{ITS}^{func} \setminus B_{ITS}^{func,M} = B_{ITS}^{func,NM}$,
$B_{ITS}^{func} \setminus B_{ITS}^{func,NM} = B_{ITS}^{func,M}$ and
$B_{ITS}^{func,NM} \cap B_{ITS}^{func,M} = \emptyset$.

The functions in ($B_{ITS}^{func,NM}$) are for general purpose: sending upper tester messages, configuring, starting, stopping and restoring the IUTs. The remaining 24 functions ($B_{ITS}^{func,M}$) are executed on *PTC R1*, *PTC R2* and *PTC M* and are describing the actual interoperability test case. The description of the dynamics of an interoperability test case (thus the behaviour of the protocol) is implemented in $B_{ITS}^{func,M}$ that only the ITS uses. Since the corresponding conformance test case checks the same test purpose, the same dynamics must be described in a conformance test case, as well. That is, the test cases of the CTS (B_{CTS}^{tc}) can be derived from $B_{ITS}^{func,NM}$.

7 Coverage of the constructed test suite

Introduce two operators: $\xrightarrow{G}$ and $\xrightarrow{R}$, in order to investigate the goodness of the CTS that we have constructed. The first one represents the process of creating a CTS from the test purposes: $TP \xrightarrow{G} CTS^G(TP)$, where TP denotes the test purposes and $CTS^G(TP)$ denotes the test suite that is based on these test purposes. This is the ordinary way to write a CTS from a textual description and considered as the best current practice. The other operator constructs a CTS from an ITS: $ITS(TP) \xrightarrow{R} CTS^R(TP)$, where $ITS(TP)$ denotes the ITS that is based on the test purposes TP. In the latter case, the two test suites (ITS and CTS^R) will be checking the same test purposes. This operator implements the method described in the previous Sections that uses an ITS to construct a CTS. The goal is to get a CTS by using $\xrightarrow{R}$ that has the same coverage as if it were created by $\xrightarrow{G}$. The test purpose TP is considered as a set that contains n elementary test purposes: $TP = (tp_1, tp_2, \ldots tp_n)$. For later use, define a test purpose TP_{prot} that contains all the elementary test purposes that can be extracted from the protocol specification.

In Section 6, we have already shown how to construct a conformance test case by re-using the corresponding interoperability test case. The resulting conformance test case checks a test purpose and it is obvious that two test cases provide the same coverage if they are both checking the same test purpose. Thus, if we want to compare the goodness of $\xrightarrow{R}$, we only have to compare the coverage of CTS^G and CTS^R. The coverage function ($Cov()$) is defined by the checked test purposes of a test suite. If test suite TS_1 and TS_2 check the test purpose TP_1 and TP_2 respectively, then we can say that the two test suites provide the same coverage:

$Cov\big(TS_1(TP_1)\big) = Cov\big(TS_2(TP_2)\big)$ if and only if $TP_1 = TP_2$.

If $TP_1 \subset TP_2$ then $Cov\big(TS_1(TP_1)\big) < Cov\big(TS_2(TP_2)\big)$. It can be seen that the coverage of a test suite depends on the set of checked test purposes (TP) and does not depend on the method the test suite was written with. Thus, if two conformance test suites check the same test purposes then they provide the same coverage:

$Cov\big(CTS^G(TP)\big) = Cov\big(CTS^R(TP)\big)$.

It is trivial that for every tp_i in $TP_{prot} = (tp_1, \ldots tp_n)$ where $1 \leq i \leq n$, a conformance test case can be written. Thus, $TP_{prot} \xrightarrow{G} CTS^G(TP_{prot})$. It is also obvious that one test case in a CTS checks only one test purpose, so the number of test cases in $CTS^G(TP_{prot})$ is n. However this is not necessarily true in case of an ITS. We have already mentioned that it is possible to write complex interoperability test cases, which check several test purposes. Since these test cases can be restructured and broken down into several test cases (where each new test case checks only one test purpose), we consider such a test case as if it were written as separate test cases. The other possibility is that a test purpose tp_i cannot be checked by an interoperability test case. For example, the answer for an erroneous or an inappropriate packet should be checked in that test purpose.

Denote TP_{CTS} and TP_{ITS} the set of test purposes that are applicable for conformance and interoperability test, respectively. Then, we can say that $TP_{CTS} = TP_{prot}$ and $TP_{ITS} \subset TP_{prot}$, what means that $TP_{ITS} \subset TP_{CTS}$, as well. If there is at least one test purpose that cannot be checked by an interoperability test case, then $TP_{CTonly} = TP_{CTS} \setminus TP_{ITS}$ and $TP_{CTonly} \neq \emptyset$. Since the ITS was written for the set of test purposes TP_{ITS}, the operator $\xrightarrow{R}$ will result in the test suite $CTS^R(TP_{ITS})$. Both $CTS^R(TP_{ITS})$ and $CTS^G(TP_{ITS})$ are based on the same test purposes, so they must provide the same coverage. That is, $Cov\big(CTS^R(TP_{ITS})\big) = Cov\big(CTS^G(TP_{ITS})\big)$. If $TP_{CTonly} = \emptyset$ then $Cov\big(CTS^R(TP_{ITS})\big) = Cov\big(CTS^G(TP_{prot})\big)$, so the test suite created by our method is just as good as if the test suite were written in the ordinary way. But, if $TP_{CTonly} \neq \emptyset$ then $Cov\big(CTS^R(TP_{ITS})\big) < Cov\big(CTS^G(TP_{prot})\big)$, what means that the test cases for test purposes TP_{CTonly} cannot be constructed from the ITS.

That is, with this method it is possible to create a CTS for TP_{ITS} test purposes and this test suite provides the same coverage as if it were written according to the current practice or in another way. The missing test cases for TP_{CTonly} must be created by some other method. Naturally, the number of such test cases depend on the protocol specification but in general it is not more than $2-5\%$ of all the test purposes. We can add these

test cases at the end of the re-use process easily because all the other test cases (and supporting templates, functions, altsteps, etc.) are already available. In our test suites, there was only one test purpose in TP_{CTonly}. According to this test purpose, if an OSPF packet is sent in which the OSPF checksum is incorrect then the IUT must not reply to it. We have written a test case by copying an already existing one that sends a correct packet and we have changed the value to an incorrect one in the checksum field. As it can be seen from this example, the construction of the missing test cases is fast and easy, since we do not have to write them from scratch.

8 An OSPF example

The OSPF protocol was developed by IETF and was published in [8]. OSPF routes IP packets based solely on the destination IP address found in the IP packet header. IP packets are routed "as is" – they are not encapsulated in any further protocol headers. OSPF is a dynamic routing protocol, it quickly detects topological changes and calculates new loop-free routes after a period of convergence. This period of convergence is short and involves a minimum of routing traffic.

In OSPF, each router maintains a database describing the network's topology. This database is referred to as the link-state database, and each participating router has an identical database. All routers run the exactly same algorithm, in parallel. From the link-state database, each router constructs a tree of shortest paths with itself as root. This shortest-path tree gives the route to each destination.

When a router starts, it uses the OSPF's Hello Protocol to acquire neighbours. The router sends Hello packets to its neighbours, and in turn receives their Hello packets. The router dynamically detects its neighbouring routers by sending its Hello packets to a special multicast address (AllSPFRouters). Each broadcast network that has at least two attached routers has a Designated Router. The Designated Router generates link-state information for the network and has other special responsibilities in the running of the protocol. This concept reduces the amount of routing protocol traffic and the size of the link-state database. The Hello Protocol also elects a Designated Router for the network. Our test suites are written for the Hello protocol.

In order to demonstrate the usage of the conversion method, we have selected the following test purpose:

```
Set different values for OSPF HelloInterval.  An incoming
Hello packet should be discarded  if the OSPF HelloInterval
does not match.
```

That is, if we send an incorrect Hello packet then the other entity will not add our IP address to his neighbour list. We test that by verifying the other party's Hello packet and if the neighbour list is empty (there are no other routers in this test) then the IUT passed this test case. The following functions are invoked in *PTC R1* in order to configure, start and stop IUT 1 (`IUT1` is a parameter that contains the attributes of IUT1):

```
IUT_Config_general ( IUT1, 20, 40, 1, IUT1.router_id );
IUT_Test_general ( IUT1 );
IUT_Stop_general ( IUT1 );
```

The very same functions are called in *PTC R2* too but with `IUT2` instead of `IUT1` and 10 instead of 20 (the second parameter of `IUT_Config_general` sets the `HelloInterval`). So, the two IUTs are configured according to the selected test purpose. The function running in the monitoring component *PTC M* checks for the following messages (the template `OSPFv2Hello_r1` only accepts Hello packets with empty neighbour list):

```
Msg1: [] OSPF.receive ( OSPFv2Hello_r1 ( p_IUT1.ip_addr,
          ALLSPFROUTERS, p_IUT1.router_id, p_IUT1.areaid ) )
Msg2: [] OSPF.receive ( OSPFv2Hello_r1 ( p_IUT2.ip_addr,
          ALLSPFROUTERS, p_IUT2.router_id, p_IUT2.areaid ) )
Msg3: [] OSPF.receive ( OSPFv2Hello_r1 ( p_IUT1.ip_addr,
          ALLSPFROUTERS, p_IUT1.router_id, p_IUT1.areaid ) )
```

If the messages are received in this order then the test is successful. The following code was cut from the constructed conformance test case to show how the same messages are handled. It can be seen too, that the very same functions are called to configure, start and stop the implementation. The code to check `Msg1` and `Msg3` can be simply copied but `Msg2` is now sent from the test suite.

```
  IUT_Config_general ( IUT, 10, 40, 1, IUT.router_id );
  IUT_Test_general ( IUT );
    ...
Msg1: [] OSPF.receive ( OSPFv2Hello_r1 ( IUT.ip_addr,
        ALLSPFROUTERS, IUT.router_id, IUT.areaid ) )
Msg2: OSPF.send ( OSPFv2Hello_helloint_s ( TESTER.ip_addr,
        ALLSPFROUTERS, TESTER.router_id, TESTER.areaid, 20 ) );
Msg3: [] OSPF.receive ( OSPFv2Hello_r1 ( IUT.ip_addr,
        ALLSPFROUTERS, IUT.router_id, IUT.areaid ) )
    ...
  IUT_Stop_general ( IUT );
```

9 Conclusion

In this paper we have presented a method that re-uses an existing ITS to produce a CTS. We have used an ITS as a base because it is much easier to write from scratch than a conventional CTS. First, we have written an ITS and a CTS for the OSPF Hello protocol. We presented how the two test suites look like and then compared them. After identifying the common parts, we have shown how to construct the CTS from the ITS. It turned out that some parts can be re-used without further modification (e.g. data type definitions, constant definitions, configuration functions). Some parts are different in the test suite, thus they must be rewritten (e.g. component type definitions). Finally, there are also some parts that can be extracted from the ITS and must be restructured to be usable in the CTS (e.g. the conformance test cases can be constructed from the interoperability "test cases"[4]).

[4]Interoperability test cases create several PTCs and the behaviour of PTCs are described in functions but we use the term "test cases" for easier understanding.

We have presented a process in Section 3 that requires the same effort to create an ITS as in the original way but then we can save considerable amount of time by re-using the existing ITS to create a CTS. At the end of this paper, we showed that the test suite created by our method covers almost the same test purposes as a CTS, which has been written by hand or by any other mean. The difference comes from the test purposes that cannot be checked by an ITS. Conformance test cases for these test purposes cannot be created by our method and must be added later but the number of such test cases are quite low.

Beside the re-using of ITS's parts, it's also an advantage that the execution environment can be used for the CTS. This is mainly done by a software module that can transmit and transform protocol messages between the test tool and the underlying service provider. The functionality of this software module depends on the TTCN–3 port types. Since the port type definitions are reusable without modification, this adaptation module is also reusable without any modification.

References

1 *OSI - Open System Interconnection, Conformance testing methodology and framework*, 1997. ISO/IEC 9646.

2 *ETSI, The Testing and Test Control Notation version 3*, August 2002.

3 **Kang S, Shin J, Kim M**, *Interoperability test suite derivation for communication protocols*, The International Journal of Computer and Telecommunications Networking **22** (March 2000), no. 3, 347-364.

4 **Kato T, Ogishi T, Shinbo H, Miyake Y, Idoue A, Suzuki K**, *Testing of Communicating Systems*: *Interoperability Testing System of TCP/IP Based Communication Systems in Operational Environment*, Ottawa, Canada, September 2000.

5 **Besse C, Cavalli A, Zadi F, Kim M**, *Testing of Communicating Systems*: *Automated Generation of Interoperability Tests*, Berlin, Germany, March 2002.

6 **Viho C, Barbin S, Tanguy L**, *Formal Techniques for Networked and Distributed Systems*: *Towards a Formal Framework for Interoperability Testing*, Cheju Island, Korea, August 2001.

7 **Dibuz S, Krémer P**, *Testing of Communicating Systems*: *Framework and Model for Automated Interoperability Test and its Application to ROHC*, Sophia Antipolis, France, May 2003.

8 **Moy J**, *OSPF Version 2, RFC 2328*, April 1998.