Testing Theory in Practice : A Simple Experiment

Rinke Terpstra, Luís Ferreira Pires, Lex Heerink, and Jan Tretmans
Faculty of Computing Science,
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{terpstra,pires,heerink,tretmans}@cs.utwente.nl

Abstract

In this paper we discuss the experiences gained in conducting a simple testing experiment. The goal of this experiment is to apply the abstract, formal testing framework "Formal Methods in Conformance Testing" [8] in a practical setting, and to indicate the critical aspects in its application to realistic testing situations. For that purpose a formal description of the system under test (SUT) is made from which tests are systematically derived. These tests are interpreted and executed by a tester against concrete implementations, thereby giving an indication about the correctness of these implementations.

1 Introduction

Conformance testing involves assessing the correctness of a concrete protocol implementation with respect to its specified behaviour by means of testing the implementation. The emerging international standard 'Formal Methods in Conformance Testing' (FMCT) [8] defines a framework on how to perform conformance testing in case the specification is described by means of formal techniques. This framework is defined at a high level of abstraction, e.g., it abstracts from specific test generation algorithms, even from a specific formal description technique. FMCT defines terminology, abstract concepts, and minimal requirements on, and relations between these concepts, such as conformance, testing, test generation, etc. Hence, use of the framework requires instantiating these concepts with specific choices for the formal description technique, for the universe of implementations, for a specific notion of conformance, for test generation algorithms, etc., while demonstrating the requirements on, and the relations between them.

The goal of this paper is to show the viability and feasibility of the FMCT framework by conducting a concrete experiment with a simple, example protocol. This experiment will show how the abstract concepts defined in FMCT (e.g., test context, points of control and observation, abstract test case) can be instantiated with concrete concepts (e.g., programming interface, executable test case). The correspondence between the abstract and concrete concepts will be investigated in order to see which concepts can be easily instantiated, and which concepts are critical in the application of FMCT to realistic testing situations.

For the experiment, a platform has been designed, in which implementations of a simple conferencing protocol entity can be tested based on a formal description of such an entity in LOTOS [5]. The protocol implementations originate from a course in Protocol Implementation [3] that is given at the University of Twente. In this course, groups of students build their implementations, independently, from an informal specification of the protocol. The implementations run in a UNIX-compatible environment, and they communicate with each other over UDP, one of the protocols in the TCP/IP protocol suite [1]. In order to test these implementations they are embedded in the distributed single layer test architecture [7]. The LOTOS simulator SMILE [2] is used to derive abstract test cases from a formal specification of the protocol in a systematic way, following the algorithm defined in [11]. The derived test cases are interpreted and executed against the implementations by a test environment, thereby giving an indication about the correctness of the implementations with respect to these tests.

In the next section we briefly recall the main concepts of the FMCT testing framework. Section 3 introduces the example conference protocol, while some implementation details of the protocol entities are described in section 4. The test architecture used for testing conference protocol implementations is elaborated in section 5. Subsequently, test generation and test execution are dealt with in sections 6 and 7, respectively, after which some concluding remarks are given in section 8.

2 Testing Framework

The emerging international standard 'Formal Methods in Conformance Testing' (FMCT) defines a framework for the use of formal methods in conformance testing [8]. This standard is intended to guide the testing process of an implementation with respect to a formal specification, and it defines, at a high level of abstraction, the concepts used in conformance testing. This section recapitulates the main concepts about conformance, testing, and conformance testing in FMCT, in so far as they are used in the subsequent sections.

2.1 Conformance

The definition of conformance concerns implementations under test (IUTs) and specifications, so a universe of implementations *IMPS*, and a universe of formal specifications *SPECS* are assumed. Implementations are concrete, informal objects, such as pieces of hardware, or pieces of software. In order to reason formally about them, it is assumed that each implementation IUT $\in$ *IMPS* can be modelled by a formal object i_{IUT} in a formalism *MODS*, which is referred to as the universe of models. This hypothesis is referred to as the *test assumption.* The test assumption only assumes that a model i_{IUT} exists; this model is not known apriori.

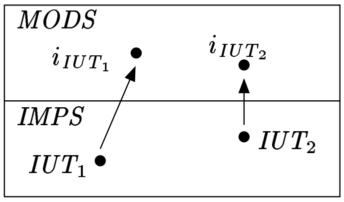


Figure 1: The test assumption visualized

Having the test assumption, which allows to reason about implementations as if they were formal objects, conformance of implementations with respect to formal specifications is expressed by means of an *implementation relation.* An implementation relation **imp** is a relation between the set of models of implementations *MODS* and the set of specifications *SPECS*: $\mathbf{imp} \subseteq \mathit{MODS} \times \mathit{SPECS}$. Implementation IUT $\in$ *IMPS* is considered **imp**-correct with respect to $s \in \mathit{SPECS}$ if and only if the model $i_{IUT} \in \mathit{MODS}$ of IUT is **imp**-related to s: i_{IUT} **imp** s. As a shorthand notation we define $M_s =_{\text{def}} \{i \in \mathit{MODS} \mid i \ \mathbf{imp}\ s\}$ as the set of models of implementations that conform to s with respect to implementation relation **imp**.

2.2 Testing

The environment in which an implementation is tested is called the *test architecture.* The test architecture gives an abstract view of how the tester communicates with the IUT. It consists of a tester, an implementation under test (IUT), a test context, points of control and observation (PCOs), and implementation access points (IAPs). A *point of control and observation* defines the communication interface between the test context and the tester. An *implementation access point* defines the communication interface between the test context and the IUT. The *test context* is the system in which the IUT is embedded, and which is there when the IUT is tested, but which is not the object of testing. The test context can be described by relating the events that occur at PCOs to the events that occur at IAPs, and vice versa. Thus, the tester communicates with the IUT via the test context (figure 2). Ideally, the IAPs and PCOs coincide, that is, the tester directly communicates with the IUT via the implementation access points of the IUT. However, practical limitations often prevent the tester from accessing the IUT directly.

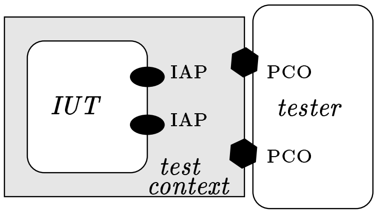


Figure 2: Test architecture

In order to formally reason about the test architecture, this architecture must be formalized in terms of its components. Following the test assumption, the IUT is formally expressed by its model i_{IUT}. The test context is modelled as a function $C : \mathit{MODS} \rightarrow \mathit{MODS}$ that transforms the behaviour of the implementation under test i_{IUT} as observed at the IAPs to the behaviour $C(i_{IUT})$ as observed at the PCOs. The PCOs and IAPs are formally modelled as interaction points in *MODS*.

The behaviour of concrete implementations is investigated by performing experiments on the implementations and observing the reactions that the implementations produce to these experiments. Such experiments are called *tests* (or *test cases*), and they are formally specified as elements of a universe of tests *TESTS*. A set of test cases $T \in \mathcal{P}(\mathit{TESTS})$ is called a *test suite* (where $\mathcal{P}(\mathit{TESTS})$ is the powerset of *TESTS*, i.e., the set of all subsets of *TESTS*). The process of running a test against a concrete implementation with the objective of investigating whether the implementation responds correctly to the test, is called *test execution.* Test execution leads to an observation in a domain of observations OBS, and is formally modelled by the function $exec : TESTS \times MODS \rightarrow OBS$. Here, the function $exec$ calculates the observation $exec(t, C(i_{IUT}))$ for implementation model i_{IUT}. This observation is made when executing test t against implementation IUT that is contained in test context C.

To each observation a verdict is assigned by a verdict assignment function $verd_t : OBS \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$, that may depend on the test t that was executed. Verdict **pass** is assigned to expected observations that may occur as a result of testing implementation IUT with test t, otherwise verdict **fail** is assigned. Implementation IUT $\in$ $IMPS$ contained in test context C passes test $t \in TESTS$ if execution of t against $C(i_{IUT})$ leads to an observation to which the verdict **pass** is assigned:

$$\text{IUT } \mathbf{passes}\ t \ \Leftrightarrow\ verd_t(exec(t, C(i_{IUT}))) = \mathbf{pass}$$

Furthermore, implementation IUT passes test suite $T \subseteq TESTS$ if IUT passes all tests in T : IUT **passes** $T =_{\text{def}} \forall t \in T$: IUT **passes** t. An implementation fails test suite T if it does not pass: IUT **fails** $T =_{\text{def}} \neg$(IUT **passes** T).

A *test purpose* describes the goal of testing with a specific test case t. Formally, this goal can be expressed as the subset of implementation models P_t that pass t (when contained in a test context C):

$$P_t =_{\text{def}} \{m \in MODS \mid verd_t(exec(t, C(m))) = \mathbf{pass}\}$$

After test execution of t against IUT we know whether IUT satisfies the test purpose of t or not, i.e., whether $i_{IUT} \in P_t$. For test suite $T \subseteq TESTS$, the test purpose P_T consists of all implementation models that pass all tests in T : $P_T =_{\text{def}} \bigcap_{t \in T} P_t$.

2.3 Conformance testing

Conformance testing involves assessing, by means of testing, whether an implementation conforms, with respect to implementation relation **imp**, to its specification. In order to judge whether a concrete implementation IUT conforms to its specification $s \in SPECS$ by means of testing, the notion of conformance, i.e., $i_{IUT} \in M_s$, and test execution, i.e., IUT **passes** T (or equivalently, $i_{IUT} \in P_T$), have to be linked, so that from test execution an indication can be obtained whether $i_{IUT} \in M_s$, i.e., whether IUT conforms to its specification. So conformance testing involves the execution of a test suite T such that the set of conforming implementations M_s is approximated by (and preferably equal to) the set of models P_T that pass test suite T.

A test suite $T \subseteq TESTS$ is complete if it can distinguish exactly between all conforming and non-conforming implementations: $M_s = P_T$. Unfortunately, this is a very strong requirement for practical testing: complete test suites are usually infinite, and consequently not practically executable. Hence, FMCT poses a weaker requirement on test suites: they should be sound, which means that all correct implementations, and possibly some incorrect implementations, will pass them: $M_s \subseteq P_T$ (figure 3(a)). In case all incorrect implementations, and possibly some correct ones, do not pass the execution of test suite T, the test suite is called exhaustive: $M_s \supseteq P_T$ (figure 3(b)).

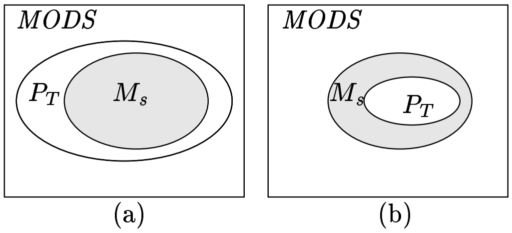


Figure 3: (a) Soundness, and (b) Exhaustiveness

The activity of deriving tests for a particular implementation relation from a (formal) specification is called *test generation.* Test generation is modelled as a function $gen^C_{\mathbf{imp}}$, that takes a specification $s \in SPECS$ and returns a test suite:

$$gen^C_{\mathbf{imp}} : SPECS \rightarrow \mathcal{P}(TESTS)$$

Following the requirement on soundness of test suites, we require that a test generation function for implementation relation **imp** produces sound test suites for any specification $s \in SPECS$:

$$M_s \subseteq P_{gen^C_{\mathbf{imp}}(s)} \tag{1}$$

3 The Conference Protocol

This section presents the example protocol: the conference protocol [3]. The service provided by this protocol, the conference service (section 3.1), the protocol data units (section 3.2.1), the underlying service used by the protocol (section 3.2.2), the behaviour of the protocol entities (section 3.2.3), and the formalization of the protocol in LOTOS (section 3.3) are all briefly discussed.

3.1 Conference service

The conference service provides a multicast service, resembling a 'chatbox', to users participating in a conference. A conference is a group of users that can exchange messages. Every user in a conference can send messages to all other conference partners participating in that conference, and it can receive messages from every other participant. The participants in a conference can change dynamically, as the conference service allows its users to join and leave a conference. Different conferences can exist at the same time, but each user can only participate in at most one conference at a time.

The conference service has the following service primitives (C-SPs), which can be performed at the conference service access points (C-SAPs):

- *join*: a user joins a named conference and defines its user title in this conference; the user title identifies a user in a conference;
- *datareq*: a user sends a message to all other users participating in the conference;
- *dataind*: a user receives a message from another user participating in the conference;
- *leave*: a user leaves the conference; since a user can only participate in one conference at a time, there is no need to identify the conference in this primitive.

The service primitives *join* and *leave* are used for conference control. The service primitives *datareq* and *dataind* are used for data transfer. Initially, a user is only allowed to perform a *join* to a conference. After this, the user is allowed to send messages, by performing *datareq*'s, or to receive messages, by performing *dataind*'s. In order to stop its participation in the conference, a user issues a *leave* at any time after it has issued a *join*.

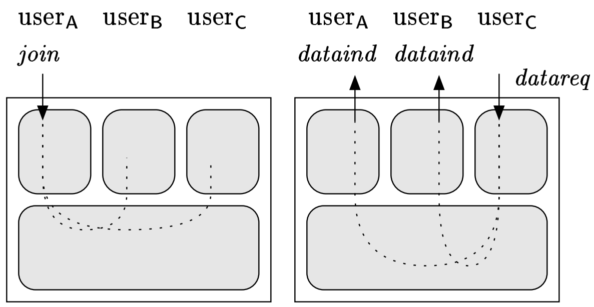
Data transfer is multicast, which means each *datareq* causes corresponding *dataind*'s in all other participants in the conference, i.e., all other users who have performed a *join* to the conference the sending user belongs to, and have not performed a *leave* after that. Data transfer in the conference service is not reliable: messages may get lost, but they never get corrupted; corrupted messages are discarded. Also, in-sequence delivery of messages is not guaranteed.

Figure 4(a) denotes the joining to a conference by $user_A$, and figure 4(b) denotes the sending of a message issued by $user_C$ to $user_A$ and $user_B$.

3.2 Conference protocol

3.2.1 Protocol data units

The conference protocol entities (CPEs) are responsible for providing the conference service. They do this by exchanging protocol data units (PDUs) via



(a) (b)

Figure 4: The conference service

an underlying service. The following PDUs are used by the CPEs:

- *join-PDU*: informs all other protocol entities that are already in the conference that the sending protocol entity has joined. This is done by sending this PDU to the set of all potential conference partners, which is established during initialization. A user title and a conference name are conveyed in this PDU;
- *answer-PDU*: answers a protocol entity that has sent a join-PDU. It is sent by all entities that are already in the conference in order to identify themselves as being a conference partner. This PDU contains the user title of the answering protocol entity;
- *data-PDU*: contains a message to be delivered to the other conference participants than the sending user;
- *leave-PDU*: informs the other conference participants that a certain participant has left the conference.

Figure 5 depicts the encoding of PDUs in octetstrings.

join	PDU type	user title	conference identifier
	00000001	10 octets	10 octets
answer	PDU type	user title	conference identifier
	00000011	10 octets	10 octets
data	PDU type	length	data
	00000100	1 octet	max 255 octets
leave	PDU type	user title	conference identifier
	00000010	10 octets	10 octets

Figure 5: Format of PDUs

3.2.2 The underlying service

The underlying service, via which the CPEs communicate with each other, is the connectionless service provided by UDP (User Datagram Protocol, [1]). UDP provides a connectionless, unreliable service: messages may get lost and may be delivered out of sequence, but they never get corrupted, nor are they misdelivered. The service primitives of the connectionless service provided by UDP can be modelled as:

- *udp_dreq (destination address, data)*: a user sends data to another user;
- *udp_dind (source address, data)*: a user receives data from another user.

Since the *udp_dind* primitive carries the source address, there was no need to include this information in the *data-PDU*.

3.2.3 Conference protocol behaviour

The CPEs are responsible for the administration of their conference partners, and for the transfer of data between conference partners. Each CPE is equipped with a set of potential conference partners, called the *configuration set*, which is established during global initialization. Such a set consists of a list of C-SAP addresses and their corresponding UDP service access points (U-SAP addresses), and it makes it possible for a CPE to reach all its peer CPEs.

Each CPE has a set of *conference partners*, which keeps track of the peer CPEs participating in the same conference. This set consists of a set of pairs, each pair consisting of a U-SAP address and a user title. Initially, when the CPE starts its operation, this set is empty.

The normal behaviour of a CPE is defined in terms of simple (informal) rules as follows; (for error situations we refer to [3]):

1. each CPE that performs a *join* primitive sends *join-PDU*s to all addresses in the configuration set. The *join-PDU* contains the user title and the conference identifier established in the *join* primitive;
2. a CPE that receives a *join-PDU* and is engaged in the conference identified in this *join-PDU* sends an *answer-PDU* to the source of the *join-PDU*, and includes the CPE that sent the *join-PDU* in its set of conference partners. The *answer-PDU* contains the user title of the user attached to the answering CPE;
3. a CPE that receives a *join-PDU* and is not engaged in the conference identified in this *join-PDU* ignores the *join-PDU*;
4. a CPE that receives an *answer-PDU* includes the address and the user title of this *answer-PDU* in its set of conference partners;
5. a CPE that performs a *datareq* sends the message of this *datareq* to all addresses of the set of conference partners;
6. a CPE that receives a *data-PDU* delivers the message contained in this *data-PDU* to its user by executing a *dataind*. The user title parameter of the *dataind* is obtained by translating the *udp_dind* source address to the user title according to the information contained in the set of conference partners;
7. a CPE that performs a *leave* sends a *leave-PDU* to all CPEs in its set of conference partners and clears this set;
8. a CPE that receives a *leave-PDU* removes the source address and the user title of the *leave-PDU* from the set of conference partners.

Figure 6(a) shows the behaviour of a protocol entity in case user_A wants to join the conference in which user_B and user_C already participate; user_A sends *join*-PDUs to both user_B and user_C, who subsequently reply with an *answer*-PDU. Figure 6(b) shows the PDUs involved when user_C sends a message to its conference participants. The instance of protocol behaviour depicted in figure 6 conforms to the instance of service behaviour depicted in figure 4. Note that it is yet to be proved that the conference protocol, defined as the composition of the conference protocol entities and the underlying service provided by UDP, indeed implements the conference service.

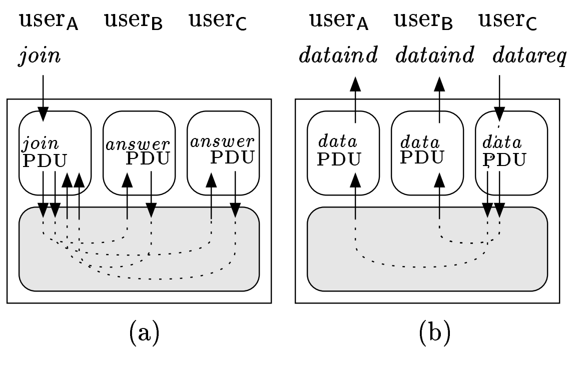


Figure 6: The conference protocol

3.3 LOTOS specification

The conference protocol was modelled in LOTOS [5] to serve as the basis for test generation in section 6. This section only gives the high-level structure of the specification; the complete specification can be found in [10].

In the conference protocol specification the service boundaries are modelled by LOTOS gates: gate `c` represents the conference service boundary, and gate `u` represents the UDP service boundary. Values of sort `CAddress` and `UDPAddress` are used at gates `c` and `u` to denote specific conference user addresses (C-SAPs) and UDP addresses (U-SAPs), respectively. These sorts are defined in the abstract data type (ADT) part of the specification. The ADT part also defines data types for configuration sets and conference-partner sets.

The high-level structure of the LOTOS specification is as follows:

```
specification ConferenceProtocol[c]
       (ConfSet:AddressPairSet) : noexit :=
hide u in
  CProtocolEntities[c,u](ConfSet)
  |[u]|
  UDPService[u]
where
  process CProtocolEntities[c,u]
      (ConfSet:AddressPairSet) : noexit :=
    choice p:AddressPair []
    [p IsIn ConfSet] -> i;
    ( CProtocolEntities[c,u](Remove(p,ConfSet))
      |||
      CProtocolEntity[c,u](p)
    )
  endproc (* CProtocolEntities *)
endspec (* ConferenceProtocol *)
```

Specification `ConferenceProtocol` represents the conference protocol, which consists of an arbitrary, finite number of CPEs (process `CProtocolEntities`) and the connectionless service provided by UDP (process `UDPService`). The number of CPEs in the LOTOS specification is determined by the number of addresses contained in `ConfSet`. The internal event `i` has been introduced in process `ConferenceProtocol` for simulation purposes, in order to avoid unguarded behaviours with infinite branching.

The connectionless, unreliable service provided by UDP is described by process `UDPService`:

```
process UDPService[u] : noexit :=
  u ?addr:UDPAddress ?sp:UDPsp [IsDataReq(sp)];
  ( i;
    ( u !DestOf(sp) !udp_dind(addr,PduOf(sp));
      stop
      ||| UDPService[u]
    )
    [] i; UDPService[u]
  )
endproc (* UDPService *)
```

4 Implementation Aspects of the Conference Protocol

This section discusses some implementation aspects of the CPE that influenced the concrete application of the formal conformance testing framework of section 2. These implementation aspects reveal details and intricacies related to the implementation platform on which the CPE was built, the configuration of the CPEs, the structure of operating system processes used, and the concrete interfaces between the CPE and its environment. This section also discusses the correspondence between the abstract events of the conference protocol specification in LOTOS and the concrete implementation constructs of the CPE implementation.

4.1 Implementation platform and structure

The conference protocol considered in this paper was implemented on SUN SPARC workstations using a UNIX-like operating system, and it was programmed using the ANSI-C programming language.

The users of the conference service are expected to be humans. This implies that a CPE implementation (our future IUT) interacts with a user interface when in operation (figure 7). This user interface can run in a different, or in the same process as the bare CPE implementation. In the former case the processes running the CPE and the user interface have to communicate using inter-process communication mechanisms; in the latter case direct procedure calls could be used for communication. We have decided to implement the CPE and the user interface as separate UNIX processes.

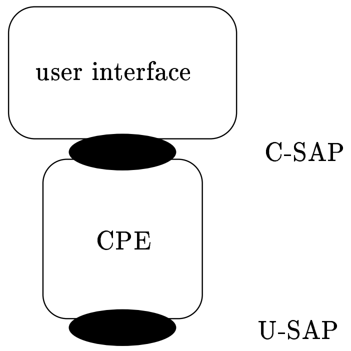


Figure 7: CPE with user interface

4.2 Configuration of the protocol entities

Each CPE must contain a *configuration set*, which enables this protocol entity to reach its peer entities.

The configuration set has to be initialized at the beginning of the CPE operation. A CPE implementation initializes its configuration set by reading an ASCII-file with a specific format, called the *configuration file*. This file contains the information necessary for setting up the configuration set.

The configuration file consists of a set of lines, one per CPE, where each line contains the conference service access point (C-SAP) identifier of the CPE, the host name of the computer system in which the CPE resides and the port number to which the CPE is attached. Each CPE also checks the consistency of the configuration file by searching and checking its own C-SAP identifier, host name, and port number in this file.

An example of configuration file is the following:

```
1   dracula     1025
2   dracula     1026
3   terminator  1025
4   terminator  1026
```

This example considers four CPE implementations, two running in host `dracula`, and two running in `terminator`. CPE implementations running in a single host are distinguished in this example by their different port numbers (1025 or 1026).

4.3 Concrete interface between CPE and user interface

The conference service interface has been implemented using *pipes*. A pipe is a uni-directional[1] communication mechanism that supports stream-oriented message exchange between processes. This means that messages sent through a pipe are delivered in order and never get lost. Bi-directional communication is achieved by a pair of pipes.

A pipe is created by using the *pipe* system call. System calls *read* and *write* enable one to receive data from, and send data to pipes, respectively. When the data transfer using a pipe is finished, the processes using this pipe have to close it using the *close* system call.

Read and write system calls can be either blocking or non-blocking. In case a read is blocking and the pipe is empty, read waits until something can be read; in the meantime it blocks the execution of the reading process. When read is non-blocking and the pipe is empty, nothing is read and the reading process continues execution. A blocking write blocks the writing process when the pipe is full. When write is non-blocking and the pipe is full, the write results in an error and the writing process continues execution.

The conference-service service primitives (C-SPs) *join*, *leave*, and *datareq* are implemented in the CPE using the *read* system call. The *dataind* C-SP is implemented using the *write* system call.

Because different types of C-SPs with different parameters are used, the CPE must be able to distinguish between the different C-SPs. The CPE deals with C-SP diversity by making use of so called interface data units (IDUs), also used in OSI terminology [4]. An IDU consists of an interface control information (ICI) part and a service data unit (SDU) part. The ICI contains the SP type and other parameters, while the SDU contains the user data. This has been implemented by defining a data structure `idu`, which is cast on the octetstrings that are exchanged with *read* and *write*. The data structure `idu` is addressed in detail in section 7.3. We have also decided that this data structure has a fixed size, and that the read and write system calls are blocking. These implementation decisions simplified considerably the programming of the interface.

The *pipe* and *close* system calls do not have any corresponding service primitive in the conference service, since initialization is not explicitly defined in our model.

4.4 Concrete interface between a CPE and UDP

A CPE communicates with other CPEs through the service provided by UDP. This means that the CPE implementation has to interact locally with the UDP implementation. We have chosen to interface with the UDP implementations through *sockets*. A socket is an abstraction that allows the generalization of the UNIX file access mechanism for interfacing with different protocols (connection-oriented, connectionless), while having different alternative roles (calling or called users).

A UDP address consists of an IP address of the host computer system and a port number. In Internet terms, a UDP port in a host identifies an *application process*, i.e., a process of the operating system in that host.

In order to be used to interface to UDP, a socket has to be created and bound to a local IP address and a port number. A socket is created using the *socket* system call, which returns, when successfully executed, a *socket descriptor*. When issuing a socket system call, the application process has to inform which protocol is going to be used. Different protocol families are supported, such as TCP/IP, Xerox and Appletalk. A socket descriptor is simply an integer value that allows an application process to refer

[1]This seems to be the most general definition of a pipe [9]. However, we noticed that pipes in the SOLARIS operating system are actually bi-directional. By considering and using pipes as uni-directional communication means we hope to achieve portability.

to the socket. A socket is bound to an IP address and a port number using the *bind* system call. After being created and bound, a socket can be used for data transfer using UDP.

The system calls normally used for sending messages using UDP are *sendto* and *sendmsg*. The system calls normally used for receiving messages using UDP are *recvfrom* and *recvmsg*. Similarly to the system calls *read* and *write*, the system calls *sendto*, *sendmsg*, *recvfrom*, and *recvmsg* can be blocking or non-blocking. In our implementation these system calls are always blocking.

When the data transfer operations using the socket are finished, e.g., when the application process is about to exit, this process is expected to close the socket using the *close* system call.

Since initialization and shutdown are not explicitly defined in the conference protocol, the *socket*, *bind*, and *close* system calls do not have corresponding service primitives in our model of the UDP connectionless service. The system calls *sendto* and *sendmsg* correspond to the *udp_dreq* service primitive. Analogously, the system calls *recvfrom* and *recvmsg* correspond to the *udp_dind* service primitive.

4.5 LOTOS and concrete interfaces

Table 1 shows the correspondence between the events of the conference protocol specification in LOTOS (section 3.3) and the implementation constructs of the conference service concrete interface. In the conference protocol specification, the C-SAP is represented by gate `c` and address `Caddr` (`c !Caddr`). The CPE uses two pipes for the implementation of the C-SAP. Pipes `Cin` and `Cout` are used for receiving and sending data, respectively. Parameter `idu` contains information on the service primitive being executed, while parameter `length` defines the length (in octets) of this information.

Table 2 shows the correspondence between the events of the conference protocol specification and the implementation constructs of the UDP service concrete interface. In the conference protocol specification, the U-SAP is represented by gate `u` and address `UDPaddr` (`u !UDPaddr`). The CPE uses a socket for the implementation of the U-SAP, which is identified by socket descriptor `sd`. Parameters *src_addr* and *dest_addr* of the *udp_dind* and *udp_dreq* SPs in the conference protocol specification correspond to arguments `src_addr` and `dest_addr` of the *recvfrom* and *sendto* system calls in the CPE implementation, respectively.

Argument `pdu` of the *recvfrom* and *sendto* system calls contain a PDU. Parameter `length` of these system calls must be equal or larger than the maximum PDU length in octets (257 octets). Since it is rather straightforward, we refrain from discussing PDU implementation in this paper.

5 Test Architecture

Section 2 introduced the ingredients of a test architecture. This section instantiates these concepts for the conference protocol, and by doing so, develops in this way a specific test architecture for this protocol. This section subsequently discusses how the implementation under test (IUT), the test context, the implementation access points (IAPs) and points of control and observation (PCOs) are implemented. This section also presents a formal model of this system, referred to as the system under test (SUT). This formal model is used in section 6 as the reference specification from which tests are derived.

5.1 Test context

In order to be able to systematically generate tests with the purpose to validate whether the behaviour of a conference protocol entity conforms to its specified behaviour, knowledge concerning the environment in which the implementation is tested, i.e., the test architecture, is essential. This requires a careful consideration of the possible ways the tester communicates with the IUT.

The most desirable option is to test conference protocol entities in isolation, i.e., to directly attach the tester to the implementation access points of the IUT. In this ideal test architecture the IAPs and PCOs coincide, and the tester is able to directly interact with the IUT (figure 8).

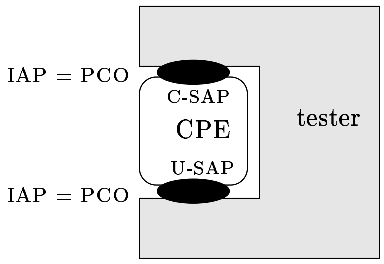


Figure 8: Ideal test architecture

Although direct interaction between the tester and the implementation under test is desirable, the ideal test architecture is difficult to realize in case a CPE implementation is tested. The reason for that is that U-SAP of the IUT is implemented using the socket communication mechanism, which prevents direct access at this U-SAP. When a tester wants to interact directly with the IUT via the U-SAP, that tester

Table 1: LOTOS events and conference service concrete interface

LOTOS event	implementation
`c !Caddr ?join(user_title, conf_id)`	`read(int Cin, char *idu, int length)`
`c !Caddr ?datareq(message)`	`read(int Cin, char *idu, int length)`
`c !Caddr ?leave()`	`read(int Cin, char *idu, int length)`
`c !Caddr !dataind(user_title, mess)`	`write(int Cout, char *idu, int length)`

Table 2: LOTOS events and conference service concrete interface

LOTOS event	implementation
`u !UDPaddr !udp_dreq(dest_addr, pdu)`	`sendto(int sd, char *pdu, int length, int flags, struct sockaddr *dest_addr, int dest_addr_len)`
`u ?UDPaddr ?udp_dind(src_addr, pdu)`	`recvfrom(int sd, char *pdu, int length, int flags, struct sockaddr *src_addr, int *src_addr_len)`

should intercept all communication from the IUT that would normally be processed by the underlying UDP layer. In this case the tester itself must emulate the underlying UDP layer. This would greatly increase the complexity of the tester itself, and is therefore not considered a realistic approach.

Another option is to adjust the IUT in such a way that other interprocess communication mechanisms can be used, so that the emulation of UDP can be omitted. This also poses difficulties. Amongst others, it requires knowledge of, and access to, the source code of the IUT. The drawback of this approach is that IUTs have to be modified, so that they are not treated as black boxes. Moreover, such modifications may change the observable behaviour of the IUT. This makes adjustment of interprocess communication mechanisms of IUTs an undesirable option.

A more realistic approach is to access the U-SAP of the implementation via the UDP layer. Now, all communication between the U-SAP of the IUT and the tester runs through the UDP layer. In ISO terminology such an architecture is called the Distributed Single Layer Testing Architecture [7] (figure 9).

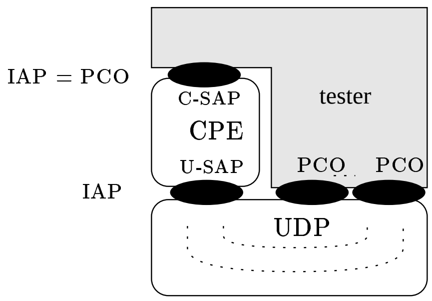


Figure 9: Distributed Single Layer test architecture

In order to test implementations of a CPE, we require that the tester and IUT reside on the same host machine. Consequently, messages exchanged between the tester and the IUT do not have to travel through the protocol layers below IP but 'bounce back' at IP. We assume that in this situation the communication between the tester and the IUT is reliable, and that messages are delivered in sequence.

Summarizing, the SUT, i.e., the system that is really tested, consists of the conference protocol entity (CPE) together with the UDP service provider. By requiring that the tester and the IUT reside on the same host machine, the UDP service can be assumed to be reliable. As indicated in section 4, the interface between the CPE and the UDP layer (U-SAP) is implemented by means of sockets, and the interface between the CPE and the tester (C-SAP) is implemented by means of pipes. In section 5.2 sockets and pipes are formally modelled, and a formal model of the SUT is constructed.

5.2 A formal model of the SUT

In order to derive tests for the SUT, the SUT has to be modelled in some formalism. Since one of its components (i.c. the CPE) has already been modelled in LOTOS, we choose to model the whole SUT as a LOTOS process. This section discusses the formal representation of IAPs and PCOs (implemented by pipes and sockets). Together with a formal model of the test context, and a formal model of the CPE, we obtain a formal description of the SUT.

Formally modelling UDP As a consequence of placing the tester and the SUT on the same host, we assume that UDP is reliable and delivers messages in sequence, in contrast with the unreliable service described by process `UDPService` (section 3). So we model the UDP service simply by a set of queues.

Interface between C-SAP and tester The PCO between the C-SAP and the tester is implemented by means of pipes, like the interface with the user interface process. A pipe essentially behaves like a bounded first-in/first-out (FIFO) buffer. A pipe is formally modelled in LOTOS by an infinite FIFO buffer under the assumption that the pipe is never "overloaded", i.e. write blocking does not occur.

Interface between U-SAP and tester The U-SAP and the interaction point between the UDP service provider and the tester are implemented by means of the datagram socket mechanism. Sockets essentially behave just like a pair of pipes, i.e., as bounded FIFO queues, and are modelled in LOTOS as infinite FIFO buffers.

All components of the test architecture have now been instantiated, and modelled. The IUT is taken as a CPE, and modelled by $i_{CPE} \in$ LOTOS. The test context is instantiated by a reliable version of UDP, and modelled by the LOTOS process `Reliable_UDP`. The IAPs and PCOs are instantiated by pipes and sockets, and modelled in LOTOS by unbounded queues. This results in a formal model of the SUT, that is described in LOTOS as follows:

```
process SystemUnderTest [UTin, UTout,
                         LTin, LTout] : noexit :=
hide Xin, Yin, Xout Yout in
  UpperTestContext[UTin, Xin, UTout, Xout]
  |[Xin, Xout]|
  ConferenceProtocolEntity[Xin, Xout,
                           Yin, Yout](cf1,udp1)
  |[Yin,Yout]|
  LowerTestContext[Yin, Yout, LTin, LTout]
endproc (* SystemUnderTest *)
```

Here, process `UpperTestContext` models the pipes used to communicate between the C-SAP and the tester, and process `LowerTestContext` models the interface between the U-SAP and the tester. A more detailed description of these processes can be found in [10].

In order to test the operational behaviour of an IUT, the tester must be able to emulate a participants involved in the same conference as the IUT. For simplicity, we fixed the number of "phantom" conference partners of the IUT to 2. Figure 10 visualizes the SUT as it is used in the remainder of this paper.

For simplicity, we optimized the SUT by replacing several consecutive unbounded queues by a single unbounded queue. This evidently does not change the overall behaviour: they all behave as a single unbounded queue.

Figure 10 shows that there is a clear distinction between input actions and output actions; an action is either an input action to the SUT, or it is an output

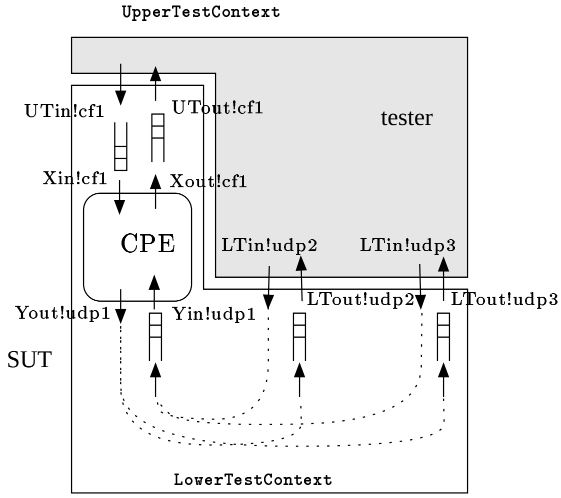


Figure 10: System Under Test

action. Table 3 denotes the input actions and the output actions, together with the interaction gates at which these actions can occur.

Furthermore, the SUT never refuses input actions; the unbounded queues are always willing to accept another input. These two properties are used in the next section, when test generation is discussed.

6 Test Generation

Before discussing test generation, we need to describe precisely how we instantiate the rest of the formal framework of section 2. This will be done in section 6.1. Then section 6.2 will briefly recall tests, test execution, and a test derivation algorithm that were introduced in [11], and section 6.3 will apply this algorithm to derive, manually, a test case from our example specification of the SUT.

6.1 Instantiating the formal framework

The formal specification of our example protocol was expressed in LOTOS in section 3.3. So, referring to the framework of section 2, we have that $SPECS =$ LOTOS. Since the semantics of any LOTOS specification is fully expressed as a labelled transition system, we could also say that $SPECS$ is instantiated with the class of labelled transition systems. A labelled transition system is a structure consisting of states with transitions, labelled with actions from an action set L, between them. We write $s \xrightarrow{a} s'$ if state s in the transition system can make a transition labelled with action $a \in L$ to state s'. We assume the standard definitions concerning labelled transition systems to be known, see e.g., [5, 11]. So,

Table 3: Input events and output events of the SUT

Input events		Output events	
SP	gate	SP	gate
join, leave, datareq	`UTin!cf1`	*dataind*	`UTout!cf1`
udp_dreq	`LTin!udp2`, `LTin!udp3`	*udp_dind*	`LTout!udp2`, `LTout!udp3`

let $\mathcal{LTS}(L)$ be the class of transition systems over action set L, then we instantiate $SPECS = \mathcal{LTS}(L)$.

The implementations that we consider are C-programs as described in section 4. Hence, *IMPS* can be considered as the class of such C-programs. The test assumption that we pose is that these C-programs behave as if they were labelled transition systems, too. Moreover, whereas LOTOS and transition systems only deal with abstract actions, without detailing whether these are inputs or outputs, we assume that these implementations do distinguish between input actions and output actions, such that input actions are initiated by, and under control of a system's environment, and outputs are initiated by, and under control of a system itself (cf. the last paragraph of section 5.2). To formalize this, we assume that models of implementations are *input-enabled*, i.e., they can always perform any input in any state. Transition systems with outputs in L_U, and with always enabled inputs in L_I are called input-output transition systems, and denoted by $\mathcal{IOTS}(L_I, L_U)$. Hence, we instantiate $MODS = \mathcal{IOTS}(L_I, LU)$. For details about input-output transition systems we refer again to [11].

The next concept that needs instantiating is the implementation relation. Here, we choose the relation **ioconf** $\subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$ of [11]. So, according to section 2.1, an IUT $\in$ *IMPS* is correct with respect to $s \in \mathcal{LTS}(L_I \cup L_U)$ if i_{IUT} **ioconf** s. The relation **ioconf** is defined by requiring that any output produced by the implementation after a sequence of actions $\sigma \in traces(s) \subseteq L^*$ (where L^* is the set of all sequences of actions in L, and $traces(s)$ is the set of sequences that can be executed by s) is also produced by the specification after the same σ, and moreover, if the implementation can have no output after σ, then the specification must also allow to have no output after σ (i **after** σ denotes the possible states in which i can be after executing σ):

$$
\begin{array}{rl}
i \ \mathbf{ioconf}\ s \ =_{\text{def}} & \\
\forall \sigma \in traces(s): & out(i\ \mathbf{after}\ \sigma) \subseteq out(s\ \mathbf{after}\ \sigma) \\
\text{and} & i\ \mathbf{after}\ \sigma\ \mathbf{refuses}\ L_U \text{ implies} \\
 & s\ \mathbf{after}\ \sigma\ \mathbf{refuses}\ L_U
\end{array}
$$

6.2 Test generation for ioconf

Before giving the test generation algorithm of [11], we first need to define what test cases are, how they are executed, and how verdicts are assigned, i.e., we need to instantiate *TESTS*, *exec*, and $verd_t$. We do this as follows.

Like specifications and implementations, tests are described as labelled transition systems. But they have some additional properties: (i) they have finite behaviour, since we are never prepared to wait infinitely long for the result of a test; (ii) they are deterministic, since, as a tester, we want full control over the testing process; and (iii) we label each state of a test case with **pass** or **fail**. Naturally, a test case intended to test an $\mathcal{IOTS}$-implementation will distinguish between inputs and outputs, where the inputs of the implementation correspond to the outputs of the test case, and vice versa. We call transition systems having the above properties input-output test cases, and we denote them by $\mathcal{IOTESTS}(L_U, L_I)$. So, we instantiate $TESTS = \mathcal{IOTESTS}(L_U, L_I)$.

A test run of a test case $t \in \mathcal{IOTESTS}(L_U, L_I)$ against an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ consists of combining the two, and having them run in parallel $(t \| i)$ until they cannot proceed any further ($(t\|i)$ **after** σ **deadlocks**). The sequence of actions which occurred until this point, $\sigma \in L^*$, is a test outcome. Because of nondeterminism in the implementation, different test runs, and hence different test outcomes, are possible. We collect all possible test outcomes into one observation. So, we instantiate $OBS = \mathcal{P}(L^*)$, and $exec : TESTS \times MODS \rightarrow OBS$ as

$$exec(t, i) =_{\text{def}} \{\sigma \in L^* \mid (t\|i)\ \mathbf{after}\ \sigma\ \mathbf{deadlocks}\ \}$$

The verdict assigned to an observation is determined by the labels of the final states of the test case after the test runs:

$$
\begin{array}{l}
\text{verd}_t(O) = \mathbf{pass} \ \Leftrightarrow \\
\forall \sigma \in O:\ t\ \mathbf{after}\ \sigma \text{ is labelled } \mathbf{pass}
\end{array}
$$

Now we have all ingredients to present the test generation algorithm of [11]. Let s be a transition system specification, let S be a set of states of s, with initially $S = \{s \mid s_0 \stackrel{\epsilon}{\Longrightarrow} s\}$, then a test case t is obtained from S by a finite number of recursive applications of one of the following three nondeterministic choices:

1. $t :=$ **stop** with label **pass** is a test case;
2. $t := a; t'$ with label **pass** is a test case if $a \in L_I$ can be executed by a state in S ($\exists s \in S : s \stackrel{a}{\Longrightarrow}$), and t' is a test case for S', which is the set of all states that can be reached via a ($S' = \{s' \mid \exists s \in S : s \stackrel{a}{\Longrightarrow} s'\}$;
3. $t := \sum\{x; t_x \mid x \in L_U\}$ is a test case if for each x that cannot be performed by an s in S ($\forall s \in S : s \stackrel{x}{\not\Longrightarrow}$) we have that $t_x =$ **stop** with label **fail**, and for each x that can be performed ($\exists s \in S : s \stackrel{x}{\Longrightarrow}$) t_x is a test case for S_x, which is the set of all states that can be reached via x ($S_x = \{s' \mid \exists s \in S : s \stackrel{x}{\Longrightarrow} s'\}$. The label of t itself is **pass** iff there is a state in S that cannot execute any output action ($\exists s \in S, \forall x \in L_U : s \stackrel{x}{\not\Longrightarrow}$).

In [11] it was proved that test suites generated by applying this algorithm are sound, i.e., satisfy (1) in section 2.3. Moreover, the test suite consisting of all test cases that can be generated with this algorithm, is exhaustive.

The attentive reader may have noticed that in the test generation algorithm the test context was not taken into account. It can be easily shown that the implementation relation **ioconf** itself is not testable for IUTs contained in the context C described in section 5: we cannot test whether i **ioconf** s if i communicates via multiple queues with its environment. The best we can do is to test for $C(i)$ **ioconf** $C(s)$, so to derive test cases from $C(s)$ to test SUTs of the form $C(i)$. This is what will be done in the next section.

6.3 A test for the conference protocol

This section describes the systematic derivation of an abstract test case from the specification of the system under test (SUT; figure 10 in section 5.2). The test case will be able to check whether a CPE at address `cf1` with user title `A`, embedded in its test context, is capable of correctly joining a conference `c`, i.e., whether it correctly responds to the action `UTin!cf1!join(A,c)`.

When we try to derive this test case by manually applying the test case generation algorithm of section 6.2, using the LOTOS simulator SMILE to generate the sequences of possible actions of the SUT, then we are overwhelmed by the complexity caused by the number of possible transitions, the number of states, etc. In order to reduce the size of the automaton representing the SUT, we use to following property, which says that if we want to have test cases for s, we can safely derive test cases from $s|[L_I]|q$ for any q with only input actions.

Proposition 1 Let $q \in \mathcal{LTS}(L_I)$, then any sound test case derived from $s|[L_I]|q$ is also sound for specification s with respect to **ioconf**. □

By choosing the right input actions in process q we restrict test generation to only these input actions. In our example we choose q = `UTin!cf1!join(A,c);stop`. This expression could represent a *test purpose* (cf. section 2.2): it describes the purpose of the test we want to generate, viz. that we want to restrict the behaviour of the SUT to the relevant part dealing with the action `UTin!cf1!join(A,c)`:

```
SystemUnderTest[UTin,UTout,LTin,LTout]
|[UTin,LTin]|
TestPurpose[UTin,LTin]

where
process TestPurpose[UTin,LTin] : noexit :=
  UTin!cf1!join(A,c) ; stop
endproc
```

Simulation of this expression using the LOTOS simulator SMILE turns out to be feasible; SMILE generates a behaviour tree representing the possible sequences of actions that the SUT can perform when `UTin!cf1!join(A,c)` is fed to the SUT. This tree is depicted in figure 11; the meaning of the numbers is explained in table 4.

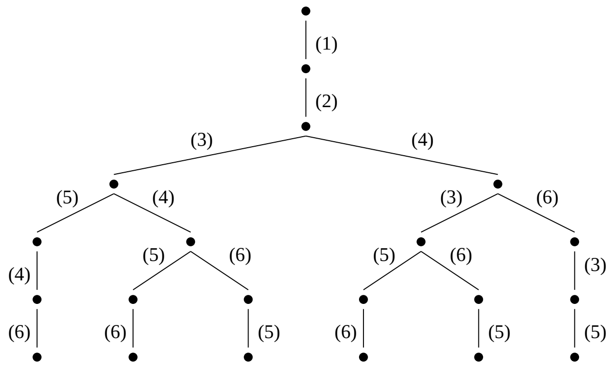


Figure 11: Simulation tree

Table 4: Label abbreviations

(1)	`UTin!cf1!join(A,c)`
(2)	`i(Xin)!cf1!join(A,c)`
(3)	`i(Yout)!udp1!udp_dreq(udp3,PDU(join,A,c))`
(4)	`i(Yout)!udp1!udp_dreq(udp2,PDU(join,A,c))`
(5)	`LTout!udp3!udp_dind(udp1,PDU(join,A,c))`
(6)	`LTout!udp2!udp_dind(udp1,PDU(join,A,c))`

The simulation tree depicted in figure 11 is the basis for applying the test generation algorithm of section 6.2, and the resulting test case is depicted in figure 12.

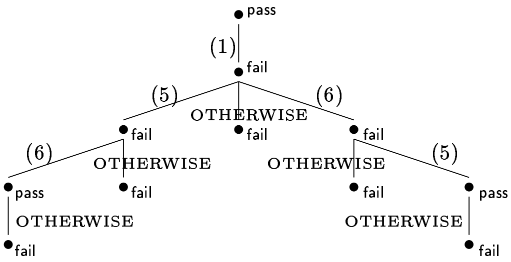


Figure 12: Abstract test case

The OTHERWISE-transitions in figure 12 represent the action that is taken when none of the alternatives applies, i.e., when the IUT produces another output then the ones explicitly specified. Figure 13 depicts the abstract test case in a TTCN-like notation (cf. [7, part 3]). Here, as it is the case in TTCN, successive events are indicated by an increasing level of indentation, and alternative events are recognized by having the same level of indentation.

```
Check if IUT correctly joins a conference
UTin!cf1!join(A,c)
  LTout?udp2?udp_dreq(udp1,PDU(join,A,c))
    LTout?udp3?udp_dreq(udp1,PDU(join,A,c))
      TIME OUT                                 pass
      OTHERWISE                                fail
    TIME OUT                                   fail
    OTHERWISE                                  fail
  LTout?udp3?udp_dreq(udp1,PDU(join,A,c))
    LTout?udp2?udp_dreq(udp1,PDU(join,A,c))
      TIME OUT                                 pass
      OTHERWISE                                fail
    TIME OUT                                   fail
    OTHERWISE                                  fail
  TIME OUT                                     fail
  OTHERWISE                                    fail
```

Figure 13: Abstract test case in test notation

The `TIME OUT`s in figure 13 correspond to the non-terminal states in the test-case tree in figure 12. They are used to detect whether or not output is generated by the IUT. No outputs is sometimes considered correct behaviour if indeed no output is expected according to the specification ("there is a state in S that cannot execute any output action" in the algorithm of section 6.2, e.g., the first `TIME OUT` in figure 13), and sometimes it leads to fail when an output was expected.

Since a LOTOS-specification cannot specify absolute timing constraints, we can never know when a message will not arrive at all, and when we simply have a very slow IUT. Hence, theoretically, the `TIME OUT`-values should be infinite, but, of course, practically this would not be acceptable. We leave the `TIME OUT`-values to the implementation and execution of the abstract test case, which will be discussed in the next section.

7 Test Execution

This section discusses how the abstract test case that was systematically generated from the LOTOS specification of the SUT (see section 6) is interpreted, translated to executable code, and executed by a tester. This implies the implementation of a tool that is able to communicate with the SUT in a way prescribed by the test case, and that is able to give a verdict about the conformance of the SUT.

7.1 Design considerations

The purpose of a tester is to perform experiments, i.e., tests, on the SUT, and decide, based on the outcome of these experiments, whether an implementation is correct or not. In the development phase of such a tester for a CPE, the following requirements were taken into account: (i) the realized tester should be constructed in such a way that it is possible to test several IUTs without having to perform major modifications in the code of the tester; (ii) the tester should be able to interpret the abstract test cases generated from the LOTOS description, and transform them into executable test cases; and (iii) the tester is specialized in testing CPE implementations in the distributed single layer testing architecture. These design decisions underlying the implementation of the tester are mainly imposed by pragmatic considerations. We chose to implement the tester in C.

In this paper the tester itself is assumed to be correctly implemented. Any mistake found in the test execution can then be considered to be caused by erroneous implementations of the CPE, and not by an erroneous implementation of the tester itself. This seems a reasonable assumption, considering that the implementation of a CPE is often more complex than the implementation of the tester.

7.2 Initialization

Before the tester can perform experiments on the SUT, i.e., interpret and execute tests, it must be initialized. Initialization involves the creation of the communication means, such as the creation of the pipes and the socket interfaces used by the tester and the IUT.

In order to create pipes between the IUT and the tester, these processes must be started up by a common parent process. The initialization (parent) process creates two pipes using the system call `pipe` (figure 14). Pipe `pipe_down` is used for communication between the tester and the IUT, and `pipe_up` is used for communication in the opposite direction. A pipe is identified by an array consisting of two pipe descriptors: the descriptor indexed by `0` is used for sending, and the descriptor indexed by `1` is used for reception. Note that pipes are bi-directional under SOLARIS. For portability reasons, however, we only will use pipes in a uni-directional fashion (see also section 4). Therefore, the non-used side of the bidirectional pipe is explicitly closed.

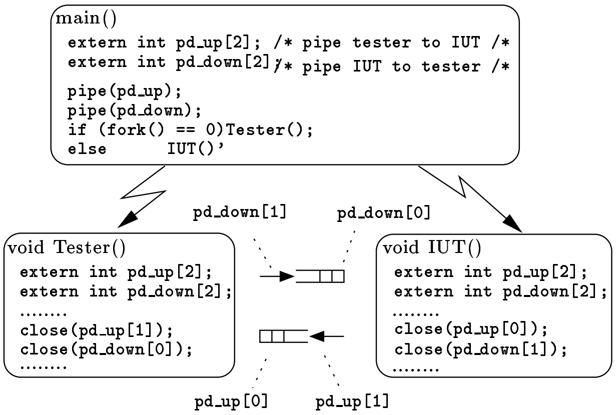


Figure 14: Initialization

For the creation of the socket interface the configuration file (see section 4) is used. In order to force the underlying UDP layer to provide reliable data transmission, it was required (section 5) that the tester and the IUT reside on the same host machine. This means that all host names occurring in the configuration file must be identical to the host name of the host where the IUT runs when the IUT is tested. As opposed to the creation of pipes, the IUT is able to create its own socket without any interference of the tester.

7.3 Transformation of test cases

Figure 13 shows a high-level notation to denote test cases. Such symbolic, abstract, TTCN-like descriptions need to be transformed into sequences of octets (octetstrings) that can be transmitted over the communication links implemented by pipes and sockets. Vice versa, octetstrings that are communicated over the communication links need to be interpreted in terms of abstract events occurring in these test cases in order to take appropriate action. This requires a mapping between the abstract events that occur in the abstract test cases, and the concrete sequences of octets which are used to represent these events, and vice versa.

The tester is responsible for the creation of the communication links between the tester and the IUT, and consequently knows the location of the PCOs (implemented by sockets and pipes) that need to be addressed in order to communicate with the IUT. Furthermore, the encoding of PDUs to concrete sequences of octets (figure 5) is part of the conference protocol specification, and thus accessible to the tester. This enables a straightforward mapping of abstract events described in an abstract test case to concrete data structures in C, and vice versa. The data structure to which all C-SPs are mapped is called an interface data unit. Figure 15 depicts the structure of an interface data unit.

```
struct idu_type {
        char type;
        char user_title[10];
        char message[255];
} idu;
```

Figure 15: Interface Data Unit

The `type` field contains the SP type, `user_title` contains the user title, and `message` field either contains the message (in case of a data SP), or the conference identifier (in case of a join SP, answer SP, or leave SP). The `idu` data structure is interpreted as a sequence of octets and transmitted over the pipe implementing the C-SAP PCO.

The tester knows how the communication links with the IUT are implemented by concrete interfaces (by means of pipes and sockets), and also knows how to address these concrete interfaces. Using tables 1 and 2, the LOTOS interactions occurring in abstract test cases are mapped to executable code in C, and vice versa. Figure 16 shows some examples of such mappings.

```
  UTin!cf1!join(A,c)
              |          idu.type = join
 mapped       |          idu.user_title = A
   to         |          idu.message = c
              v
  write(pipe_down[1],(char *)&idu,sizeof(idu))


  read(pipe_up[0],(char *)&idu,sizeof(idu))
              |          /* idu.type == dataind */
 mapped       |          /* idu.user_title == B */
   to         |          /* idu.message == c */
              v          pipe_up[0] corresponds with UTout!cf1
  UTout!cf1?dataind(B,c)
```

Figure 16: Transformation of events

For example, in figure 16 the abstract event `UTin!cf1!join(A,c)`, generated by the tester in order to check whether user `A` can successfully join conference `c`, is implemented by the C code as indicated in the `write` statement. Here, the abstract interaction point `UTin!cf1` is mapped to a concrete pipe descriptor `pipe_down[1]` that is used to implement this interaction point.

7.4 Interpretation and execution of abstract test cases

In order to interpret and execute the abstract test cases an algorithm is designed and implemented. The core of this algorithm is described in pseudo-code in figure 17.

manoeuvre tester to initial test behaviour
while (*no verdict received*)
  read test alternatives
  if (*alternative is an input action*)
  then if (*new* PCO $\neq$ *old* PCO)
        then *wait certain period*
        fi
        translate abstract input expression
        perform concrete input expression
        increase level of alternatives
  else *wait for event to occur*
        if (*event* = `TIME OUT`)
        then *determine verdict*
        else *read received data from buffer*
                translate data to abstract event
                compare translated data with alternatives
                if (*translated data is a service primitive*)
                then *increase level of alternatives*
                else /* `OTHERWISE` *or* `ELSE` */
                        determine verdict
                fi
        fi
  fi
endwhile

Figure 17: Algorithm for test case execution

The tester starts by reading all immediate alternative actions offered by the abstract test case. From section 6 we know that test cases are prepared either to accept all output actions (from the perspective of the SUT), or to perform a single input to the SUT. Since the tester interprets abstract test cases that, by definition, comply to these requirements, it suffices for the tester to consider the acceptance of all outputs, and the sending of only one input.

If the tester performs an input action to the SUT, a concrete octetstring has to be transmitted via a communication buffer, indicated by the PCO of the input action. If this PCO equals the communication buffer used to send an octetstring the last time, then in-sequence delivery is guaranteed, since the buffers are assumed to be FIFO buffers (section 5.2). However, if octetstrings are sent via different buffers, it may be possible that one overtakes the other. In order to increase the probability of in-sequence delivery of data in this case, we let the tester wait for a certain period of time.

If the tester expects an output action from the SUT, it waits for the reception of an octetstring. If such reception does not occur in time, a `TIME OUT` event is generated, and the corresponding verdict is assigned to the outcome of the test case. Upon the reception of such an octetstring, this string is interpreted and translated to an event.

We have assumed that octetstrings are never corrupted, and thus always correspond to a valid abstract event. Depending on the received abstract event, corresponding action will be taken; either a verdict is established, or a next test event is executed.

8 Concluding Remarks

We discussed in this paper how to instantiate the abstract, formal testing framework of "Formal Methods in Conformance Testing" [8] for testing a concrete, simple, example protocol. The instantiation turned out to be feasible, although some critical aspects have been identified.

Most intricacies are found in the test context, i.e., in the communication between the IUT and the tester. Every component of the test context must be faithfully modelled. Also, the means of communication, like pipes and sockets, must be faithfully included. Such means of communication behave as buffers, so they can delay messages, and if two such buffers are put in parallel, the order of messages sent by the tester need not be preserved when they arrive at the IUT.

The IUT and whole test context was formally described, and this served as the basis for the formal generation of test cases. So, test cases were generated for the system under test (SUT), i.e., the IUT together with the test context, and this was the system that was actually tested. It is not possible to test the IUT in isolation due to the test context. The test context decreases the testability of the IUT.

The abstract events of the specification had to be mapped onto concrete structures, like octetstrings, and vice versa. This mapping is important, and needs to be clearly described. We did this by hand, but a challenge would be to see to what extent this can be automated. The relation with de/encoders that deal with standard encodings of abstract types, such as ASN.1/BER, needs further investigation [6].

The test generation algorithm turned out to be applicable, however, it appeared to be already rather complex for this relatively simple example protocol.

We only derived a couple of very simple test cases. Better tool support is indispensable in this respect.

Although timing constraints cannot be specified in the specification formalism at hand, they turn out to be important when testing concrete implementations. First, a time-out value needs to be specified after which we can be sure that no message will ever arrive. This is necessary to conclude when an IUT does not produce an (expected) output. Secondly, sometimes it is convenient for the tester to wait for some time with sending an input to the IUT, in order to be sure that the IUT received and processed the previous input, especially if the two inputs are sent via different buffers (pipes or sockets) to the IUT.

The tester that we developed is not as general as we had hoped it to be. Actually, until now, we only tested one implementation with it (which produced the expected outcomes). For testing another IUT it might be that it has to be changed to cope with other concrete means of communication, most notably the communication between the bare protocol entity (the IUT) and the user interface. A general tester is only possible if also the concrete interfaces at programming level of the IUT are standardized.

References

[1] Douglas E. Comer. *Internetworking with TCP/IP. Volume I; Principles, Protocols and Architecture.* Prentice-Hall International, Inc., USA, 2nd edition, 1991.

[2] H. Eertink. Executing LOTOS specifications: The SMILE tool. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 221–234. Kluwer Academic Publishers, 1995.

[3] L. Ferreira Pires. Protocol implementation: Manual for practical exercises 1995/1996. Lecture notes, University of Twente, Enschede, The Netherlands, August 1995.

[4] ISO. *Information Processing Systems, Open Systems Interconnection, Basic Reference Model.* International Standard IS-7498. ISO, Geneve, 1984.

[5] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* International Standard IS-8807. ISO, Geneve, 1989.

[6] ISO. *Information Processing Systems, Open Systems Interconnection, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* International Standard IS-8825. ISO, Geneve, 1990.

[7] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework.* International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[8] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Proposed ITU-T Z.500 and Committee Draft on "Formal Methods in Conformance Testing".* CD 13245-1. ISO – ITU-T, Geneve, 1996.

[9] Marc J. Rochkind. *Advanced UNIX programming.* Prentice-Hall Software Series. Prentice-Hall Inc, New Jersey, USA, 1985.

[10] H.F. Terpstra. Application of testing theory in a practical setting and realization of a tester for the conference protocol. Master's thesis, University of Twente, Enschede, The Netherlands, 1996.

[11] J. Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 127–146. Lecture Notes in Computer Science 1055, Springer-Verlag, 1996.