

# Development of a Tool to Test Computer Protocols

W.D. Myburgh



THESIS PRESENTED IN PARTIAL FULFILMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE  
AT THE UNIVERSITY OF STELLENBOSCH.

Supervised by: Prof. P.J.A. de Villiers

April 2003

# Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature:

Date:

# Abstract

Software testing tools simplify and automate the menial work associated with testing. Moreover, for complex concurrent software such as computer protocols, testing tools allow testing on an abstract level that is independent of specific implementations. Standard conformance testing methodologies and a number of testing tools are commercially available, but detailed descriptions of the *implementation* of such testing tools are not widely available.

This thesis investigates the development of a tool for automated protocol testing in the *ETH Oberon* development environment. The need to develop a protocol testing tool that automates the *execution* of specified test cases was identified in collaboration with a local company that develops protocols in the programming language *Oberon*. Oberon is a strongly typed *secure* language that supports modularisation and promotes a readable programming style. The required tool should translate specified test cases into executable test code supported by a runtime environment. A *test case* consists of a sequence of input actions to which the software under test is expected to respond by executing observable output actions.

A number of issues are considered of which the first is concerned with the *representation* of test case specifications. For this, a notation was used that is basically a subset of the test specification language *TTCN-3* as standardised by the *European Telecommunications Standards Institute*.

The second issue is the format of *executable* test cases and a suitable *runtime environment*. A translator was developed that generates executable Oberon code from specified test cases. The compiled test code is supported by a runtime library, which is part of the tool. Due to the concurrent nature of a protocol environment, concurrent processes in the runtime environment are identified. Since *ETH Oberon* supports multitasking in a limited sense, test cases are executed as cooperating background tasks.

The third issue is concerned with the *interaction* between an executing test case and a system under test. It is addressed by an implementation dependent interface that maps specified test

interactions onto real interactions as required by the *test context* in which an implementation under test operates. A supporting protocol to access the service boundary of an implementation under test remotely and underlying protocol service providers are part of a test context.

The *ETH Oberon* system provides a platform that simplifies the implementation of protocol test systems, due to its size and simple task mechanism. Operating system functionality considered as essential is pointed out in general terms since other systems could be used to support such testing tools. In conclusion, directions for future work are proposed.

# Opsomming

Toetsstelsels vir programmatuur vereenvoudig en outomatiseer die slaafse werk wat met toetsing assosieer word. 'n Toetsstelsel laat verder toe dat komplekse gelyklopende programmatuur, soos rekenaarprotokolle, op 'n abstrakte vlak getoets word, wat onafhanklik van spesifieke implementasies is. Daar bestaan standaard metodes vir konformeringstoetsing en 'n aantal toetsstelsels is kommersiëel beskikbaar. Uitvoerige beskrywings van die implementering van sulke stelsels is egter nie algemeen beskikbaar nie.

Hierdie tesis ondersoek die ontwikkeling van 'n stelsel vir outomatiese toetsing van protokolle in die ontwikkelingsomgewing van *ETH Oberon*. Die behoefté om 'n protokoltoetsstelsel te ontwikkel, wat die *uitvoering* van gespesifiseerde toetsgevalle outomatiseer, is geïdentifiseer in oorleg met 'n plaaslike maatskappy wat protokolle ontwikkel in die *Oberon* programmeertaal. Oberon is 'n sterkgetypeerde taal wat modularisering ondersteun en a leesbare programmeerstyl bevorder. Die toetsstelsel moet gespesifiseerde toetsgevalle vertaal na uitvoerbare toetskode wat ondersteun word deur 'n looptydomgewing. 'n Toetsgeval bestaan uit 'n reeks van toevoeraksies waarop verwag word dat die programmatuur wat getoets word, sal reageer deur die uitvoering van afvoeraksies wat waargeneem kan word.

'n Aantal kwessies word aangeraak, waarvan die eerste te make het met die voorstelling van die spesifikasie van toetsgevalle. Hiervoor is 'n notasie gebruik wat in wese 'n subversameling van die toetsspesifikasietaal TTCN-3 is. TTCN-3 is gestandardiseer deur die *European Telecommunications Standards Institute*.

Die tweede kwessie is die formaat van uitvoerbare toetsgevalle en 'n gesikte looptydomgewing. 'n Vertaler is ontwikkel wat uitvoerbare Oberon-kode genereer vanaf gespesifiseerde toetsgevalle. Die vertaalde toetskode word ondersteun deur 'n biblioteek van looptydfunksies, wat deel van die stelsel is. As gevolg van die eienskap dat 'n protokolomgewing uit gelyklopende prosesse bestaan, word daar verskillende tipes van gelyklopende prosesse in 'n protokoltoetsstelsel geïdentifiseer. Aangesien *ETH Oberon* 'n beperkte multitaakstelsel is, word toetsgevalle

vertaal na eindige outomate wat uitgevoer word as samewerkende agtergrondtake.

Die derde kwessie het te make met die interaksie tussen 'n toetsgeval wat uitgevoer word en die stelsel wat getoets word. Dit word aangespreek deur 'n koppelvlak wat gespesifiseerde interaksies afbeeld op werklike interaksies soos vereis deur die konteks waarin 'n implementasie onderworpe aan toetsing uitvoer. 'n Ondersteunende protokol om die dienskoppelvlak van die implementasie oor 'n afstand te bereik en ander onderliggende protokoldienste is deel van 'n toetskonteks.

Die *ETH Oberon*-stelsel help in die vereenvoudiging van die implementasie van protokol toetssstelsels, as gevolg van die stelsel se grootte en die eenvoudige taakhanteerder. Die essensiële funksionaliteit van bedryfsstelsels word uitgelig in algemene terme omdat ander stelsels gebruik kan word om toetsstelsels te ondersteun. Ten slotte word voorstelle vir opvolgwerk gemaak.

# Acknowledgements

I want to thank my parents, brother and sister for encouragement, and especially my supervisor, Pieter de Villiers, for his guidance in my studies.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Opsomming</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Project Motivation . . . . .	2
1.2 Project Goal . . . . .	2
1.3 Thesis Outline . . . . .	3
<b>2 An Overview of Protocol Testing</b>	<b>4</b>
2.1 Software Testing . . . . .	4
2.2 Protocol Specifications and Implementations . . . . .	6
2.2.1 Message types and contents . . . . .	7
2.2.2 Orderly exchange of messages . . . . .	8
2.2.3 Protocol Implementations . . . . .	9
2.3 Conformance Testing . . . . .	10
2.3.1 Abstract Test Architecture . . . . .	10
2.3.2 OSI Conformance Test Architectures . . . . .	11

2.3.3	White Box Testing . . . . .	13
2.4	The Process of Protocol Testing . . . . .	13
2.4.1	Test Generation . . . . .	14
2.4.2	Notations for Abstract Test Suites . . . . .	16
2.4.3	Test Execution . . . . .	20
2.5	Protocol Testing Tools . . . . .	20
2.5.1	Telelogic Tau TTCN Suite . . . . .	21
2.5.2	OpenTTCN Tester . . . . .	22
2.5.3	Testing Technologies's TT tool series . . . . .	22
2.6	Summary . . . . .	23
<b>3</b>	<b>Test System Requirements and TTCN-3</b>	<b>24</b>
3.1	Representation of Test Specifications . . . . .	26
3.1.1	CSP as a Test Specification Language . . . . .	27
3.1.2	An Overview of TTCN-3 . . . . .	29
3.1.3	Remarks . . . . .	36
3.2	Executable Test Suites and Runtime Support . . . . .	36
3.2.1	Communication . . . . .	37
3.2.2	Timers . . . . .	37
3.2.3	Control and Test Case Execution . . . . .	38
3.2.4	A TTCN-3 Test System . . . . .	39
3.3	Real Interaction with an IUT . . . . .	40
3.4	Summary . . . . .	42
<b>4</b>	<b>Implementation of a Tool for Protocol Testing</b>	<b>43</b>
4.1	The ETH Oberon System . . . . .	44

4.2 A Test System Overview . . . . .	46
4.3 A Test Specification Language and its Translation . . . . .	48
4.3.1 A Test Suite . . . . .	48
4.3.2 Test System Interface . . . . .	49
4.3.3 Basic Types . . . . .	50
4.3.4 Structured Types and Message Types . . . . .	51
4.3.5 Message Templates . . . . .	53
4.3.6 Test Case Behaviour . . . . .	55
4.4 A Runtime Environment . . . . .	57
4.4.1 A Communication Framework . . . . .	57
4.4.2 Runtime Support . . . . .	60
4.5 Interaction with an Implementation under Test . . . . .	67
4.5.1 The Structure of a Test Adapter . . . . .	68
4.5.2 An Adaptation in the System under Test . . . . .	71
4.6 Remarks . . . . .	72
<b>5 Conclusion</b>	<b>74</b>
5.1 Future Work . . . . .	76
<b>A List of Abbreviations</b>	<b>77</b>
<b>B Syntax of Test Specification Language</b>	<b>79</b>
B.1 Declarations . . . . .	79
B.2 Event Statements . . . . .	80
B.3 Expressions . . . . .	81
B.4 Tokens . . . . .	81

B.5	Comments . . . . .	82
B.6	Predefined Type Names . . . . .	82
<b>C</b>	<b>Test Suite Example</b>	<b>83</b>
C.1	Generated Module for Type Declarations . . . . .	87
C.2	Generated Module for Test Suite Behaviour . . . . .	89
<b>D</b>	<b>The PTSRuntime Module Definition</b>	<b>95</b>
D.1	Exported Declarations . . . . .	95
D.2	Runtime Operations for Test Execution . . . . .	96
D.3	Procedures used by a Test Adapter . . . . .	96

# Chapter 1

## Introduction

Computer systems are key components in the digital telecommunication world of today. They communicate with one another on a network by using wireline or wireless network technology and well defined protocols that determine how the communication should take place. For example, the TCP/IP protocol suite is used in data communications [42]. Protocol implementations are usually complex software systems and must be tested to ensure conformance to their appropriate specifications.

Protocol entities in computers are by nature reactive systems, which means that they continually receive various types of input and produce a wide range of output. These interactions happen at two distinct boundaries: at the *upper boundary* (or *service boundary*) interaction takes place between the user of the protocol services and the protocol implementation; at the *lower boundary* interaction takes place between the protocol and its communication environment or service provider. Sometimes, however, it is not possible to make this distinction, for example in routing protocols there are only lower boundaries.

Protocol implementations are tested by executing *test cases*. A test case describes interaction events and the data associated with them. Each test case is specified according to a *test purpose* derived from the protocol specification, to check certain specified properties of the implementation under test (IUT). Interactions can be stimulus events, to which an IUT is supposed to react, or can be response events, which are observed as *output* produced by an IUT. More than one possible sequence of interactions may be allowed in a test case. For example timeout events or the occurrence of specific alternative responses from the IUT may result in different sequences of interactions. Responses that are not allowed always result in a *failed* test case.

## 1.1 Project Motivation

The need to develop a tool to test protocol implementations was identified in collaboration with a local company that develops protocols in the *Oberon* programming language. Protocol tests used by the company were usually handwritten Oberon code. Hence, testing involved spending time on *how* some aspect should be tested rather than on *what* should be tested.

Oberon is a small but powerful programming language designed by Niklaus Wirth [35]. It is a strongly typed *secure* language that supports modularisation and promotes a simple and readable programming style. An ETH Oberon system that runs on an Intel processor, called *Native Oberon*, contains a basic TCP/IP stack implementation and is being used as a development environment for protocol implementations.

In my search for published experiences on the implementation of protocol testing tools, I found that such publications are not readily available. The standardization authorities are aware of the advantages of these tools, because there is an on-going process of standardisation of conformance test methodologies and test case representations [12, 20, 33]. Currently there is much discussion on the recently standardised test specification language *TTCN-3* (Testing and Test Control Notation version 3) [9, 10]. Several commercial and in-house tools for protocol test systems have been developed [25, 29, 44, 46]. Most of these implementations are proprietary. However, one of the vendors of TTCN-3 development tools, Testing Technologies [46], recently has made some of their source code available subject to a licence agreement.

## 1.2 Project Goal

The main goal of this project was to develop a protocol testing tool for automatic test execution for the ETH Native Oberon system. The tool allows the specification of a suite of test cases in an appropriate notation that can be translated to an executable form to realize automated test execution.

Since no protocol testing tool existed for the ETH Oberon development environment, it was decided to develop a prototype of such a tool on a Native Oberon machine that included an *Ethernet* adapter, which can be connected to a target machine hosting the IUT. An Ethernet connection is assumed to be reliable with a short cable between the two machines. To illustrate how the test system can be used to test protocol implementations, the *Internet Protocol* (IP) and *Address Resolution Protocol* (ARP) implementations of Native Oberon are

used in examples.

### 1.3 Thesis Outline

Software testing in general and protocol conformance testing are introduced in Chapter 2.

A number of general requirements for a protocol testing tool are stated in Chapter 3. Requirements to be satisfied by test systems and test specification languages in general, with TTCN-3 as a guideline, are presented. Particular attention is given to a number of issues, namely

- the representation of test case specifications,
- the format of executable test cases,
- a suitable runtime environment to support execution of test cases, and
- the interaction between an executing test case and an implementation under test.

Chapter 4 describes the implementation of an experimental protocol testing tool in Native Oberon. A test specification language is used that is basically a subset of TTCN-3. The translation of a test suite specification to Oberon is described. Test cases are translated in the form of finite state machines, which are executed as cooperating background tasks in the Native Oberon system. The runtime environment provides operational support for the execution of test cases and a communication framework to adapt communication events for a particular test context.

In Chapter 5 a number of important issues regarding test systems are summarized.

## Chapter 2

# An Overview of Protocol Testing

In the early eighties a foundation was laid for protocol test systems by D. Rayner [32]. He discussed a development plan for a test system that could test OSI (Open Systems Interconnection) protocol implementations. Later in that decade, he led the standardisation of the OSI Conformance Testing Methodology and Framework (CTMF) at the International Standards Organisation (ISO) [33]. Although CTMF is aimed at OSI protocols, it provides useful guidelines and can be used as a starting point for the testing of protocols in general.

This chapter starts with an introduction to software testing in general. Section 2.2 presents an overview of protocol specifications and implementations. OSI conformance testing concepts are introduced in Section 2.3. Protocol testing with the appropriate theory is presented in Section 2.4. Section 2.5 concludes this chapter with a brief overview of existing protocol testing tools.

### 2.1 Software Testing

The goal of software testing is to determine whether a software implementation works as expected or not. In other words, software is tested to demonstrate correct execution or to find *failures* in particular scenarios. However, it is not possible to demonstrate correct execution in general. Software testing as such is not concerned with the detection of the *cause* of failure. It is rather concerned with the detection of the presence or absence of failures. When it is found that a failure has occurred, the *fault* (or *bug*) in the executed code that caused the failure has to be located to apply the appropriate correcting measures. A *debugger* is used to locate faults in program code.

The process of software testing involves the execution of the software code with appropriate sets of *input* data so that the resulting *output* can be observed to make a judgement on the correctness of software behaviour. A description of preconditions, input values, expected output and postconditions is called a *test case*. A set of test cases is grouped into a *test suite*.

During development, software systems are sometimes built in incremental steps with the addition of extra functionality at each step. To ensure that additions do not affect previous builds, previous tests must be repeated. This process is called *regression testing*. Three levels of software testing that correspond to different levels of abstraction have been identified in [23]: *Unit testing* is concerned with the testing of individual functions in a *unit* (or a group of coherent functions) at the lowest level; *Integration testing* has to do with the testing of the integrated use of those functions, that is the combined use of units or modules; and in *system testing* the specified functional requirements of an entire software system or sub-system are tested.

There are two fundamental approaches to derive test cases [23]:

1. *Black box testing* or *Functional testing*: A software implementation is seen as a “black box” so that no insight into the implementation is used. Only a reference specification is used to derive test cases that test functional requirements of the implementation.
2. *White box testing* or *Structural testing*: The internal structure of the implementation is used for the derivation of test cases. Structural coverage metrics such as statement coverage, branch coverage and multiple condition coverage are used to determine how thorough an implementation is tested by a set of test cases.

A *test coverage analyser* is a testing tool that shows the coverage of an executed test suite in terms of coverage metrics. Such a tool was implemented for Oberon [27]. It instruments the given source code by inserting counters, which are used to generate a coverage report after the execution of a set of test cases. This tool can determine coverage up to a level of *multiple condition coverage*, which means that every simple boolean expression in the given source code is tested for both outcomes [23].

Test cases can be divided into the following kinds of tests, where each kind is trying to answer a different question [47]:

- *Functional tests* or *conformance tests*: Does the system comply with its functional specification?

- *Performance tests*: Does the system work as fast as required? What is its responsiveness under various conditions?
- *Robustness tests*: How does the system react if its environment shows unexpected behaviour? How well does the system recover from various error conditions and abnormal situations?
- *Stress testing*: Can the system cope with heavy loads, for example when its environment is delivering requests at a very high rate? What is its throughput?
- *Reliability tests*: How long can we rely on the correct functioning of the system? What is the mean time between failure?

The main approach followed in the literature to test protocol implementations, however, is functional testing, also known as *conformance testing* [32, 40, 47]. Other types of protocol testing include performance testing, robustness testing, and interoperability testing [7, 36]. Interoperability testing determines whether two (or more) implementations will communicate with each other correctly.

## 2.2 Protocol Specifications and Implementations

A communication protocol is a set of rules that govern an *orderly exchange* of messages of *particular types* among communicating entities [39]. For each message, type rules are specified for the *format* it should have as well as for the detailed *encoding*, which is the representation when it is transmitted through a computer network.

An exchange of messages can happen between peer protocol entities as for example between two TCP entities on different machines. These messages are called *protocol data units* (PDUs). Protocol entities in adjacent layers, for example a TCP entity and an IP entity on the same machine, communicate via appropriate *service primitives* (SPs). In practice PDUs are sent via a service primitive of an underlying protocol entity. Service primitives may be implemented as procedure calls or with appropriate inter-process communication mechanisms of an operating system. Figure 1 shows this exchange of messages. A protocol entity has two distinctive interface boundaries: An *upper service boundary* for the exchange of service primitives and a *lower boundary* for the exchange of protocol data units.

According to the definition above, protocol specifications can be divided into two parts: A description of message types and the data contents they should transfer, and a control part

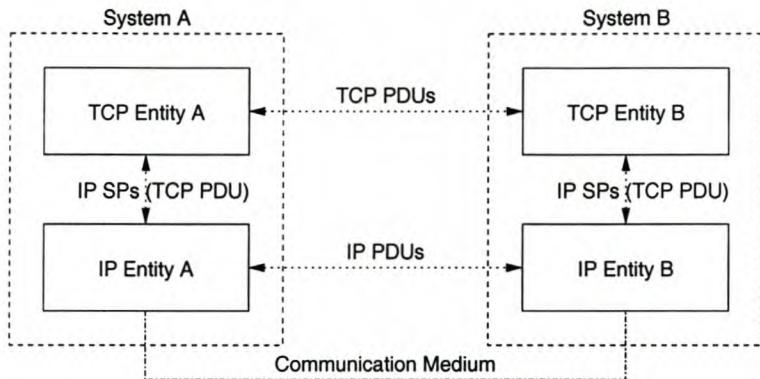


Figure 1: Peer protocol entities exchange PDUs and adjacent protocol entities exchange SPs.

which describes procedures to ensure the orderly exchange of messages. These parts are discussed in the next two subsections.

### 2.2.1 Message types and contents

The protocol data units and service primitives consist of data *fields* and *parameters*, respectively, which may influence the actions taken by the control part of a protocol. PDUs are part of a transmission message on a communication medium. It is therefore required that their formats must be specified precisely for an unambiguous representation during a transmission. PDUs can be defined in two ways:

- A *bit string or binary form* describes the number of bits used per field in the correct order of transmission. The order of transmission is usually resolved to the octet level. This style is preferred for protocols that use a few types of protocol data units with fixed types of content. It should be clearly specified how *numeric quantities* are encoded (that is in *big endian* or *little endian* order). Lower layer protocol data units such as IP datagrams and TCP segments are specified in this format.
- An *abstract syntax* can be used in conjunction with encoding rules to transform a message into the appropriate *transfer syntax*, which is the binary representation during transmission. ASN.1 (Abstract Syntax Notation One) is the standardised notation for this purpose [28]. Encoding rules such as the *Basic Encoding Rules* (BER) exist for ASN.1 [39]. Higher layer PDUs such as those of the *Simple Network Management Protocol* (SNMP) are specified with ASN.1 [41]. Table 1 presents some of the simple types and structured type notations. A “SEQUENCE” in ASN.1 corresponds to a

ASN.1 Simple Type	Denoted set of values
BOOLEAN	Truth values: TRUE, FALSE.
INTEGER	The whole numbers.
REAL	The real numbers.
BIT STRING	Sequences of zero or more bits.
OCTET STRING	Sequences of zero or more arbitrary octets.
ASN.1 Structured Type	Denoted set of values
SEQUENCE { <i>TypeList</i> }	Sequences of <i>named</i> elements which correspond to the types in <i>TypeList</i> .
SEQUENCE OF <i>Type</i>	Sequences of zero or more <i>anonymous</i> elements of the type given by <i>Type</i> .

Table 1: Some of the simple types and structured types in ASN.1

```
ExamplePDU ::= SEQUENCE {
    address  INTEGER (0..127),
    flag      BOOLEAN OPTIONAL,
    data      OCTET STRING
}
```

Figure 2: A simple PDU specified in ASN.1

record or “struct” in a programming language, while a “SEQUENCE OF” corresponds to an array of elements of a specific type. An example of a simple PDU described in ASN.1 is shown in Figure 2.

The encodings of service primitives are not important in specifications because they only have meaning inside a computer system. They are usually only specified in terms of abstract data types [17].

### 2.2.2 Orderly exchange of messages

The control part of a protocol specification describes the actions to ensure that the correct messages are sent in the correct order and with the correct data. Sometimes a distinction is made between a control component and a data component. The *control component* only describes a *Finite State Machine* (FSM) stating what type of message should be sent in a specific state in response to an internal system event (such as a *timeout*) or an external event (such as an incoming message). These states are associated with specific *phases* in the operation of a protocol entity, for example a connection establishment phase, a data transfer

phase, and a disconnection phase. Actions that manipulate the data associated with messages and internal data structures are described in the *data component*. The establishment of connections, data transfer mechanisms, error processing, acknowledgement schemes and flow control are some aspects that are described in this component.

Internet protocol specifications, such as RFCs (*Request for Comments*) are only described textually in an informal natural language [38]. The cause of many problems encountered during testing today is indeed the unclear and ambiguous descriptions in these specifications [40, 47]. However, when protocol specifications are described by means of formal techniques a formal framework for testing can be used to generate tests. The advantage of formal specifications is that they can logically be verified to have certain behavioural properties. For example, a model checker can be used to “*test*” whether a formal specification satisfies the required properties [19]. This “*test*” is done on all the executable paths of a formal specification. In contrast to formal verification, testing can only be done on a subset of all the execution paths.

### Formal Specifications

Different formal specification languages have been developed such as the standardised Estelle, LOTOS and SDL [26]. Other languages also in use are Promela [19] and CSP [18]. CSP is a *process algebra*. The main goal of process algebras is the formal specification of process behaviours on a high level of abstraction. These algebras define transformation rules and equivalence relations for the formal reasoning about behaviours. LOTOS and Promela are CSP-like languages that describe behaviour in terms of sequential *processes* which can take part in sequences of *events*. Processes can be combined to execute in parallel and may synchronise with one another in specialised *communication events*: output in CSP is denoted by “ $c!v$ ” where  $c$  is the name of a channel on which communication takes place and  $v$  is the value of the message which is sent; similarly, input is denoted by “ $c?x$ ” where  $x$  is a variable storing any value that is received on the channel.

#### 2.2.3 Protocol Implementations

A protocol is implemented by transforming a specification into an executable form by means of a programming language. This process may be error-prone when it is done manually; here the need for protocol testing arises. The difficulty, however, lies in the possibility that these translations have to make a shift between a process-oriented language and an imperative programming language. This problem might be overcome if a process-oriented programming

language is used. When a protocol has a formal specification, implementations can be obtained semi-automatically [36]. Testing also plays an important role in the formal specification context to verify a significant subset of all the executable paths when a formal verification technique is inappropriate.

## 2.3 Conformance Testing

Protocol conformance testing involves the testing of protocol implementations to check that a protocol implementation conforms to its specification. Two alternative approaches in conformance testing are described in the literature:

- In *passive testing*, traces of inputs and outputs exchanged between an *operating* implementation and its operational environment are observed and collected [43]. The traces are then verified whether they are valid according to the specification. No active tester is used to drive the implementation.
- In *active conformance testing*, an implementation is placed in a *dedicated test environment*. According to predefined test cases, the test execution system drives the implementation with specified input interactions and observes and analyses the outputs by comparing them with expected output.

The idea behind passive testing can also be used in active testing as illustrated in [50] where trace analysis is used in conjunction with a test execution system.

### 2.3.1 Abstract Test Architecture

A description of the environment in which an implementation is tested, is called a *test architecture*. This standardised term is defined in ITU-T Recommendation Z.500 [22]. An abstract test architecture is shown in Figure 3. It consists of a *tester*, an *implementation under test* (IUT), a *test context*, *points of control and observation* (PCOs), and *implementation access points* (IAPs).

The tester is the test case executor and evaluator. It provides communication interfaces to the test context. These interfaces are defined by points of control and observation (PCOs). PCOs are used to *control* the actions that are applied to the IUT and to *observe* the responses from the IUT via the test context. The test context is the environment (or system) in which

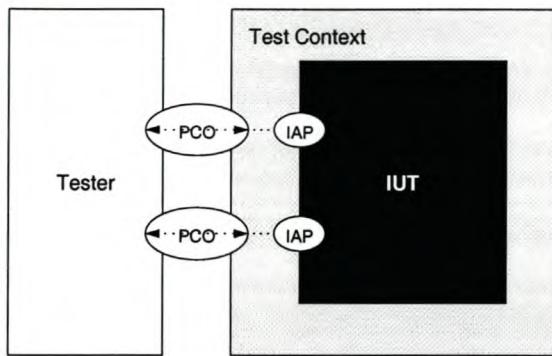


Figure 3: Abstract Test Architecture

the IUT is embedded. It is not part of the object of testing (the IUT) but is only used by the tester to access the IUT. An IAP defines the communication interface between the test context and the IUT. This usually corresponds to a programming interface or inter-process communication primitives of an operating system. The test context relates the events that occur at PCOs in the communication between the test context and the tester, to events that occur at IAPs in the communication between the test context and the IUT. Thus, the tester communicates with the IUT via the test context.

It would be ideal if the tester could communicate directly with the IUT, i.e., when the IAPs and PCOs coincide. However, this is usually impossible because of practical constraints as mentioned in [45]:

- The tester and IUT may reside on different machines;
- IAPs are sometimes only accessible indirectly, for example via an underlying *service provider*;
- IAPs are not always standardised interfaces that could be implemented by the tester. An intermediate mapping is therefore necessary.

### 2.3.2 OSI Conformance Test Architectures

OSI protocol test architectures can be described in terms of protocol layering as proposed in the OSI CTMF (Conformance Testing Methodology and Framework) standard (ITU-T Recommendation X.290 [20]): the tester can conceptually be divided into an *Upper Tester* that interacts with the upper boundary of the IUT, and a *Lower Tester* that interacts with

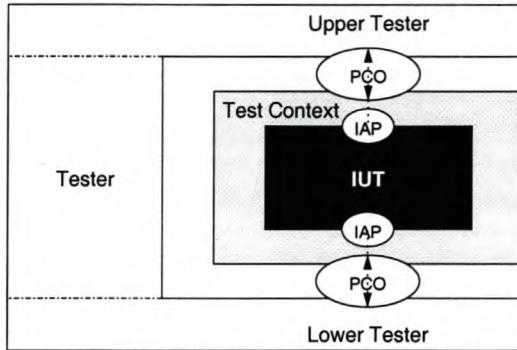


Figure 4: A conceptual architecture for protocol testing

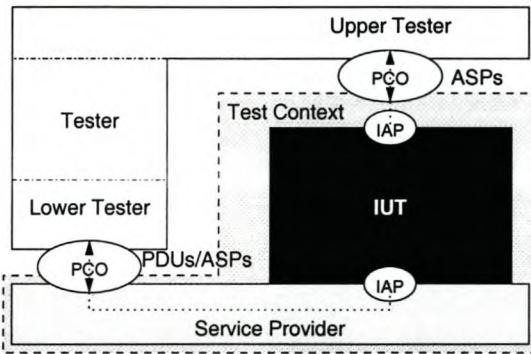


Figure 5: A test architecture where a service provider is part of the test context

the lower boundary of the IUT. The conceptual architecture is illustrated in Figure 4.

In practice the test context below the IUT consists of an underlying *service provider* as shown in Figure 5. The PCO at the lower tester provides the interface between the tester and the service provider to access the lower boundary of the IUT. This architecture may be realized on a single machine when the IUT lies above a protocol service provider that supports communication between entities on the same machine. In addition, the tester must also be supported by the same operating system. Otherwise the tester and IUT may be supported by two machines.

The OSI CTMF distinguishes between four types of test architectures with the use of a service provider, related to four test methods, namely the *local*, *distributed*, *coordinated* and *remote* test methods [20]. These methods differ in the number and positions of PCOs in a test architecture.

A protocol implementation is usually supported by suitable system services. These services, however, are not tested directly and are therefore assumed to be reliable. These services include timers, memory management and underlying protocol services. A protocol IUT might also become part of the system services when higher layer protocol implementations are tested. The complete system in which an IUT is operational is called a *system under test* (SUT). A SUT is therefore part of the test context.

The architectures described so far are aimed at *single-party testing*, which describes the context where the IUT only communicates with a single peer protocol entity. In cases where it is possible for the IUT to communicate with more than one entity concurrently, it might be necessary to extend these architectures to cater for *multi-party testing*. In this case, multiple test components are necessary and it is possible that they could be executed in a *distributed test system*. The synchronisation among parallel testers is one of the problems encountered in distributed testing [2].

### 2.3.3 White Box Testing

Conformance testing can be combined with white box testing: the source code of an implementation is *instrumented* with probes that could report internal events and variable values to a test system. It can be used as diagnostics that indicate which part of an implementation is responsible for faulty behaviour [7, page 1837]. Care should be taken not to change the intended behaviour of the implementation. In this context white box testing is a valuable aid during debugging.

It can also be used to report internal path traces which could be used directly by specific test cases. White box testing is often necessary in the case of reactive systems [30, page 60]. For this kind of testing, additional *internal* access points must be created inside the implementation to be (controlled and) observed by the tester.

## 2.4 The Process of Protocol Testing

There are two major phases in the active conformance testing process: *test generation* and *test execution* [47]. Test generation involves the selection or derivation of abstract test cases from a specification, as discussed in Section 2.4.1. The OSI Conformance Testing Methodology and Framework (CTMF) uses the terms *abstract* and *executable* to distinguish between implementation independent and implementation dependent concepts respectively [37]. A

collection of abstract test cases in an *abstract test suite* is thus written independently of the test execution system and system under test. Standardised notations exist to describe abstract test suites, which are discussed in Section 2.4.2. When an abstract test suite must be executed, an *executable test suite* is generated that is executable on a test system, which is connected to a SUT. An overview on test execution is given in Section 2.4.3.

### 2.4.1 Test Generation

Several conformance test generation methods have been developed [3, 4, 7, 48]. The automation of test suite generation is still an active research area that relies heavily on formal specifications of software and their underlying semantic models.

The FSM model, more specifically an Input/Output FSM, is the basic model used in test generation [14]. Some of the best known methods for this model are *transition tours*, the *Unique-Input-Output* method, and the *Distinguishing Sequence* method. One drawback is that these models can test only the control component of a protocol — no data dependencies are taken into account.

A more sophisticated model is the Extended FSM [7]. The FSM model is extended with *parameters* for input and output interactions and *variables*. Each transition is associated with an *enabling predicate* in terms of the current values of the variables and the actual parameters of an input action. A transition is also associated with an action on the variables and an output action to be executed provided that the enabling predicate is true. This model underlies some specification languages such as Estelle and SDL. The basic problem found with this model in test generation is that to find a path for which all transitions are executable, is known to be undecidable [7]. Such a path is needed to produce an event sequence for a test case. However, techniques based on control and data flow heuristics, can be used to select test sequences [7, 49].

Another model that is frequently used with specification languages such as CSP, LOTOS and Promela is that of *labelled transition systems* [6, 39, 48]. Tretmans presented two nondeterministic algorithms to generate test cases based on this model randomly and on-the-fly in [48]. In *on-the-fly* conformance testing test primitives are derived from a specification during test execution as a test run progresses [6].

According to OSI CTMF a test case is generated with a well defined *test purpose* in mind [20, 21]. A test purpose defines the requirements in a specification that should be tested. A typical test case, based on an FSM model for the implementation, has the following structure:

1. A *preamble* puts the IUT in the required state in which the expected behaviour corresponding to the test purpose can be observed.
2. A *test body* initiates the appropriate actions to check the observed reactions from the IUT against the test purpose.
3. A *verification step* verifies that the IUT is in the expected state.
4. A *post-amble* puts the IUT back into a neutral state.

Every path (or test outcome) of a test case also must result in the assignment of a *test verdict* to a test case. A test verdict can be one of three alternatives:

- *Pass*: The observed test outcome conformed to the requirements of the test purpose, and is a valid sequence of events according to the specification.
- *Fail*: The observed test outcome did not conform to the requirements of the test purpose, or invalid test events were observed.
- *Inconclusive*: The test outcome was a valid sequence of events, but the test purpose could not be satisfied.

Apart from the proper assignment of test verdicts, a test case should properly terminate. In other words every path defined in a test case should be finite.

### Manual Test Suite Production

The specification of protocol test suites is still done manually in many cases and sometimes semi-automatically with the use of interactive test generation tools [3]. An example is the SAMSTAG method where test purposes are defined with Message Sequence Charts, which are used to generate test cases from SDL specifications [16]. Test cases derived manually can also have faults, but they can be formally validated against a formal specification to have the functional properties of their test purposes. The validation of proper termination and verdict assignments can also be performed [24].

### 2.4.2 Notations for Abstract Test Suites

The ISO standardised notation for the specification of OSI conformance test suites is called *Tree and Tabular Combined Notation* or TTCN [12, 31]. It is being used to specify conformance test cases that can be expressed abstractly in terms of the control and observation of protocol data units (PDUs) and abstract service primitives (ASPs).

As the name suggests, TTCN consists of a combination of two types of notations: a tree notation and a tabular one. The *tree notation* is used in the dynamic test behaviour descriptions to describe alternative test events (branches from a node in a tree), which can occur after a previous event (root node of a subtree). All static elements, such as data types, PDU and ASP definitions, are declared in the *tabular notation*. There are also tables that can be used to specify ASPs and PDUs with ASN.1. A BNF syntax description of a TTCN also exists for machine processing, but is not used as the main notation, because it is not easily readable by humans [12].

#### Scenario for a Test Case Example

Consider the following scenario:

When an IP (*Internet Protocol*) user requests the sending of an IP packet to some destination, the host IP entity determines through a routing algorithm what the following intermediate destination in the network is on a path to the final destination. Assume that the final destination is a different host on the same subnetwork and that *Ethernet* is the underlying technology. First, if the Ethernet address of the destination is not in the cache, it must be resolved with ARP (*Address Resolution Protocol*). An *ARP-request* with the IP address of the destination is broadcasted on the subnetwork. When the destination host receives the request, it replies with an *ARP-reply* that includes its Ethernet address. After the requesting host has received the reply, it sends the required IP packet to the destination at the replied Ethernet address.

A test case example called “*ArpIpSend*” emulates the IP user and the destination of an IP packet in this scenario. The IUT (implementation under test) consists of an ARP and IP implementation. This example in TTCN is shown in Figure 6. It represents a typical behaviour description in a TTCN table that consists of the following columns:

Test Case Dynamic Behaviour					
		<b>Test Case Name:</b> ArplpSend <b>Group:</b> <b>Purpose:</b> <b>Configuration:</b> <b>Default:</b> <b>Comments:</b> Final Destination is on the same subnetwork; Ethernet is the underlying technology <b>Selection Ref:</b> <b>Description:</b> The sending of an IP packet with the use of an ARP request			
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		UTPCO!IPSend START ReceiveTimer			
2		LTPCO?ArpPacket			
3		LTPCO!ArpPacket			
4		LTPCO?IPPacket CANCEL ReceiveTimer			
5		?TIMEOUT ReceiveTimer			
6		LTPCO?IPPacket CANCEL ReceiveTimer	IpPacket0	PASS	
7		?TIMEOUT ReceiveTimer	IpPacket0	FAIL	

Figure 6: A test case in a TTCN table

- The *Nr* column indicates row numbers;
- The *Label* column allows to specify labels for TTCN statements that are referenced by GOTO statements;
- The *Behaviour Description* column is used to define TTCN statements. Statements on the same indentation level indicate alternatives, i.e., branches in the behaviour tree, while successive statements on incrementing indentation levels indicate a statement sequence. Send and receive statements are denoted by “!” and “?” respectively. In this example two defined PCOs (points of control and observation), namely UTPCO and LTPCO, for the upper and lower tester respectively are referenced in the communication events. The test case is started with an IPSend primitive that is sent to the upper boundary of the IUT (row 1). A timer, ReceiveTimer, is then started. One of the subsequent alternatives (in rows 2, 6, or 7) must happen: if no specified ArpPacket message (row 2) or IPPacket message (row 6) is received from the lower boundary within the time specified by ReceiveTimer, a timeout event will occur (row 7). However, when an ArpPacket message with contents specified by ArpRequest is received (row 2), the test case replies with an ArpPacket message specified by ArpReply (row 3). The test case then waits for a IPPacket message specified by IPPacket0 (row 4) or a timeout (row 5).

- The *Constraint Ref* column provides references to constraints, which specify concrete values or value ranges of PDUs (or ASPs) that should be sent or received. In this example `IpPacket0` specifies the value content of the PDU with type `IPPacket` that could be received in row 4 or 6.
- The *Verdict* column includes verdict assignments to indicate the success or failure of a test run with respect to the sequence of statements that have been performed. In both cases where a timeout may occur in the example, a `FAIL` verdict is assigned. However, when the specified `IPPacket` has been received, a `PASS` verdict is assigned to the test case.

The `ArpIpSend` test case example will be used throughout this thesis to show different notations and to explain some issues.

## TTCN-2

Two additions have been made in the second edition of TTCN. These are concurrency and modularisation:

- *Concurrent TTCN* allows for the expression of test behaviour in a distributed way. The need of specifying concurrent test case components arises in a multi-party test context. A test case is managed by a so-called *main test component* (MTC), which is responsible for the main test case behaviour and the creation of the necessary *parallel test components* (PTCs). A MTC also issues the final verdict, depending on the verdicts received from the PTCs. Synchronisation between test components is managed by the exchange of *coordination messages* (CMs) at *coordination points* (CPs) as opposed to ASPs and PDUs, which are exchanged at PCOs.
- *Modular TTCN* added the concepts of reuse and sharing into TTCN. Many protocols are composites of other smaller protocols and therefore have a natural modular relationship. Tests for these protocols can be described in separate modules by the use of the *import* and *export* tables.

One drawback of TTCN is that its language attributes are specific to OSI conformance testing, which restricts its use in other testing environments.

### TTCN-3

Recently, the *European Telecommunications Standards Institute* (ETSI) developed a more generic solution by the introduction of a complete new test specification language called TTCN-3. Refer to the ETSI ES 201 873-1 standard for the complete language definition [10]. An overview of a subset of the language is given in Section 3.1.2. TTCN-3 is intended to be used in different kinds of software testing including protocol testing, unit testing, interface testing, integration testing, as well as the testing of distributed systems. (Currently, extensions for the testing of real-time reactive systems are investigated [5].) The name was changed to *Testing and Test Control Notation version 3* [9] and its syntax is similar to C or Java.

```
testcase ArpIpSend() runs on ArpIpTestComponent {
    // The sending of an IP packet after a possible ARP request

    timer ReceiveTimer := 1.0; // one second

    UTPCO.send(IpSendRequest);
    ReceiveTimer.start;
    alt {
        [ ] LTPCO.receive(ArpRequest) {
            LTPCO.send(ArpReply);
            alt {
                [ ] LTPCO.receive(IpPacket0) {
                    ReceiveTimer.stop;
                    setverdict(pass);
                }
                [ ] ReceiveTimer.timeout {
                    setverdict(fail);
                }
            }
        }
        [ ] LTPCO.receive(IpPacket0) {
            ReceiveTimer.stop;
            setverdict(pass);
        }
        [ ] ReceiveTimer.timeout {
            setverdict(fail);
        }
    }
}
```

Figure 7: A test case in TTCN-3

The “*ArpIpSend*” example in TTCN-3 is shown in Figure 7. The test case behaviour is exactly the same as in the case of TTCN. The test case *ArpIpSend()* runs on a test component of the type *ArpIpTestComponent*, which in this case corresponds to the test system interface. A test system interface is regarded as the collection of communication ports that are connected to the IUT, like PCOs in TTCN. The communication ports *LTPCO* and *UTPCO* are declared in the *ArpIpTestComponent* type definition. A timer *ReceiveTimer* is declared with a default duration of one second. Communication events are specified with **send** and **receive** statements associated with communication ports, and alternatives are explicitly specified in

an **alt** construct. A **setverdict** statement allows the assignment of test verdicts.

### Message Sequence Charts

It is also possible to specify an abstract test case graphically with a *Message Sequence Chart* (MSC). A graphical presentation format for TTCN-3 called GFT is based on MSCs [1]. Refer to the ETSI ES 201 873-3 standard for a complete description [11]. Figure 8 shows the “ArpIpSend” example in GFT. The topmost horizontal line with an arrow on the right side of the vertical line in the middle indicates that a message of type `IPSend` is sent at the port `UTPCO` and the message contents is defined by `IpSendRequest`. After that the timer `ReceiveTimer` is started. Alternative sequences are separated by the dashed lines in the blocks as indicated by a keyword **alt** in their left top corners. The behaviour further on is the same as described before.

#### 2.4.3 Test Execution

Test execution is implemented by test suites, which can be compiled and executed on a physical machine or interpreted by a virtual machine. Usually, a runtime system supports these executions and the means to connect an IUT with the test program are provided. Test verdicts for each test case are analysed automatically so that an overall conformance report can be generated. The test program usually records the test events so that they can be inspected after testing.

A wealth of material on experiences regarding the implementation of automatic protocol test execution tools is not available. Most testing tools are implemented by commercial vendors who keep their implementations proprietary. These tools are usually the result of research and as such are being used as research tools. Discussions between tool vendors and researchers in standardisation bodies are continually taking place. Consequently, automatic test execution is further investigated in the next two chapters.

## 2.5 Protocol Testing Tools

Examples of commercial protocol testing tools are OpenTTCN Tester [29], Telelogic TAU TTCN Suite [44], and the TT tool series of Testing Technologies [46]. A brief overview of these tools is provided in the following subsections.

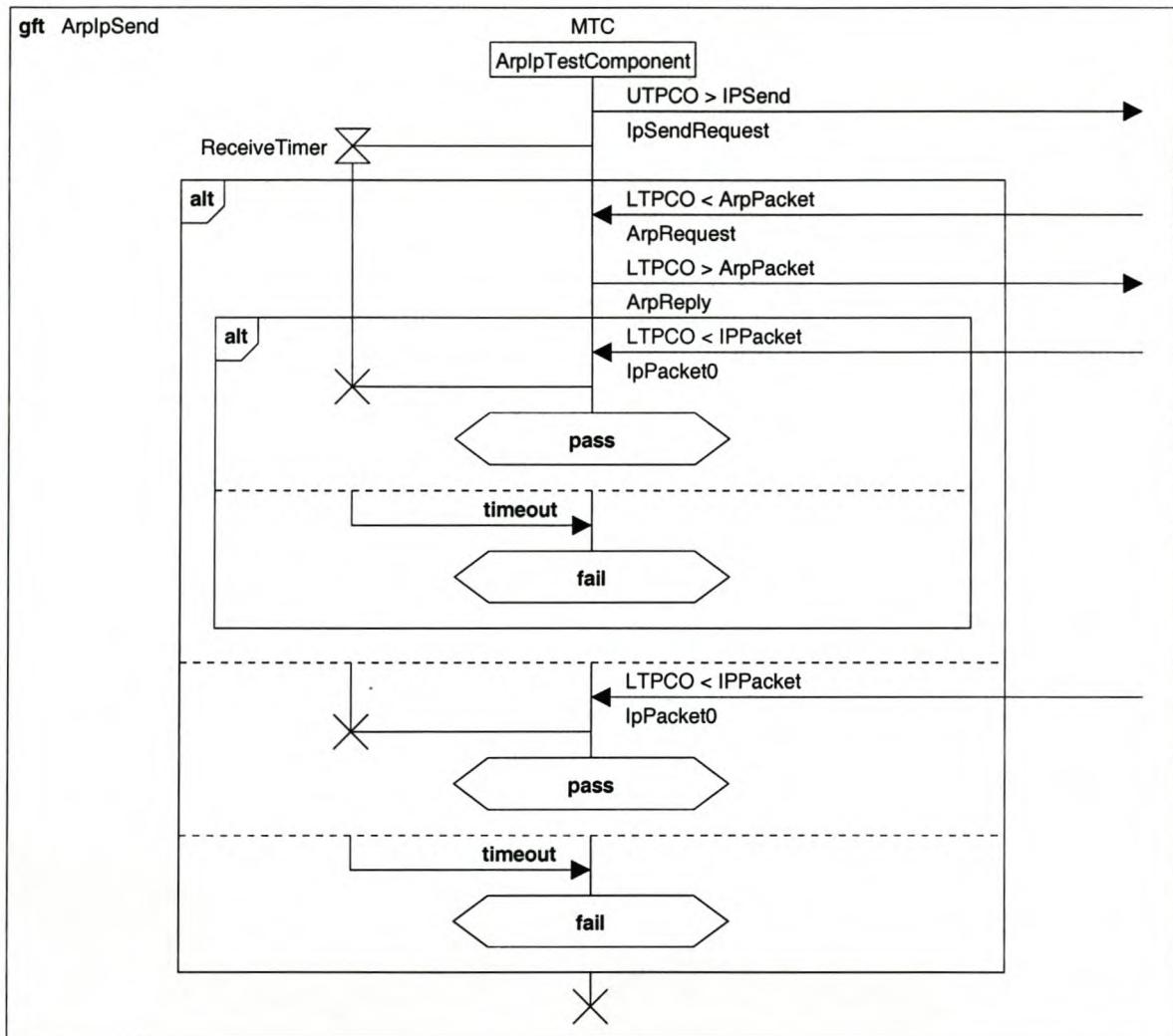


Figure 8: A test case represented in GFT

### 2.5.1 Telelogic Tau TTCN Suite

The *Telelogic Tau TTCN Suite*, previously known as ITEX (Interactive TTCN Editor and eXecutor), of the Swedish company *Telelogic AB* provides a conformance testing environment with TTCN-2 as test suite language on different Windows and Unix platforms [44]. It is part of a development environment for advanced software systems, called *Telelogic Tau*. It provides integrated design, simulation, testing, and implementation debugging tools. It also includes automated code generators translating specifications into executable code.

Communication software systems can be specified in SDL with the *Telelogic Tau SDL Suite*. Support for MSC (Message Sequence Charts) is also included. A possible behaviour sequence described in MSC can be checked against an SDL specification to be valid. These MSC's can

then be used to generate test cases with the use of an automatic test generation tool, called AUTOLINK.

The TTCN Suite compiler translates a TTCN test suite into ANSI C code, which is independent of the target operating system and protocol. It is expected that extra code is provided with a standardised interface (called GCI or *Generic Compiler/Interpreter Interface*) to the produced C-code, to realize a test run with the intended implementation under test [25].

Telelogic also introduced an automated TTCN-3 testing tool, called *Telelogic Tau/Tester* recently.

### 2.5.2 OpenTTCN Tester

The *OpenTTCN Workstation Tester*, from *Open Environment Software* in Finland, is running on a Linux system [29]. The *OpenTTCN Engine* interprets and executes abstract test cases in TTCN-2 and TTCN-3 with its virtual machine. The CORBA architecture is employed to connect a SUT to the test server. An *Interface Server Library* for C++ and Java is provided as an application programming interface on top of CORBA for the implementation of so-called Interface Servers.

An Interface Server is implemented with the instantiation of objects of the provided *PCO class*, which is registered in the test server as a point of control and observation. A PCO instance must provide SUT-specific procedures to send and receive PDUs, via an appropriate protocol stack, to and from the implementation under test. The CORBA architecture makes it possible to implement Interface Servers on any platform that supports it, for example Windows, UNIX systems, and Java machines.

### 2.5.3 Testing Technologies's TT tool series

Testing Technologies, situated in Berlin, Germany, provides testing tools for TTCN-3 [46]. Their products consist, amongst others, of a graphical test development tool for the graphical format of TTCN-3, a TTCN-3 to Java compiler, and a test execution and control tool, which are called *TTspec*, *TTthree*, and *TTman* respectively. These tools were implemented in Java and run on Windows and Unix platforms with a Java Runtime Environment.

## 2.6 Summary

In this chapter a broad overview of protocol testing was given. Protocol implementations are tested according to a protocol specification by using black-box conformance testing techniques. Conformance Test Architectures were introduced, and the protocol testing process was discussed in terms of test specification or generation and test execution. The standardised OSI test suite notation TTCN (and TTCN-2) and the recently standardised TTCN-3 for general test specifications were introduced with the same example of a test case in each notation.

As mentioned in Chapter 1 the need for a protocol testing tool for the ETH Native Oberon system was identified. By following the discussion on the development of TTCN-3 and its Runtime Interface, TRI [13], much insight was gained on the internal requirements of a test runtime environment and a tool necessary to automate the execution process.

The next chapter outlines the general requirements of a tool for automated protocol test execution. Some design issues are also highlighted in Chapter 3. In chapter 4 the focus will be on the development of a minimal prototype for the Native Oberon environment.

## Chapter 3

# Test System Requirements and TTCN-3

In this chapter general requirements for protocol test systems are discussed. The test specification language TTCN-3 is explained in the light of these requirements. Tools are often used to ease specific tasks that are tedious or inherently difficult. In protocol testing, writing test programs for protocol implementations is such a task. Test cases are derived from the design specification of an implementation that must be tested. To find a set of test cases with an acceptable coverage is not a trivial task. Manual development of test cases is widely used, but automatic generation of test cases is an active area of research. The use of a test coverage analyser can help in the development of a set of test cases with an acceptable coverage.

In contrast to user interface testing, where a human could provide input to test a program, protocols are reactive systems that do not have a user interface. Instead, computer programs interact with protocol implementations, locally and remotely. Protocol tests could be developed as test programs in a programming language. The problem is that when test cases are developed in this way too much effort is spent on *how* a test case is executed rather than on *what* should be tested.

One way to address the problem is to provide a library of abstractions that can be used for test specifications in a programming language. However, with this approach some execution issues might still affect how test cases are developed, for example the use of concurrency mechanisms specific to an underlying operating system. An alternative is to use a special *test specification language* that allows *abstract* test specifications. With this approach, test cases are specified independent of a test execution environment while the focus remains on

the specification of an implementation to be tested (i.e., a black box testing approach). A test specification is also independent of an implementation. A test specification, for example, would specify the emission of a service primitive without the knowledge of what procedure calls or interprocess communication is required by an implementation.

When it is decided to use a test specification language, a separate *translation process* would be concerned with how test case specifications are executed on a specific test machine. A test specification must be translated to an executable format that may either be interpreted or executed. In addition to this dynamic and test specific part of a test system, a test specification language should be supported by a runtime environment. The choice of operating system and the existing runtime support for specific programming languages determine how this issue is addressed.

A test specification and its translation are not necessarily concerned with *real interaction* between an executing test case and an implementation under test, especially when the tool is aimed to be as general as possible. With a general (abstract) approach, a complete executable test suite (a translated test suite specification) would still be independent of an implementation under test and its test context. An *interface* is thus required for a mapping between communication events in a test case and real interactions with an implementation under test (IUT).

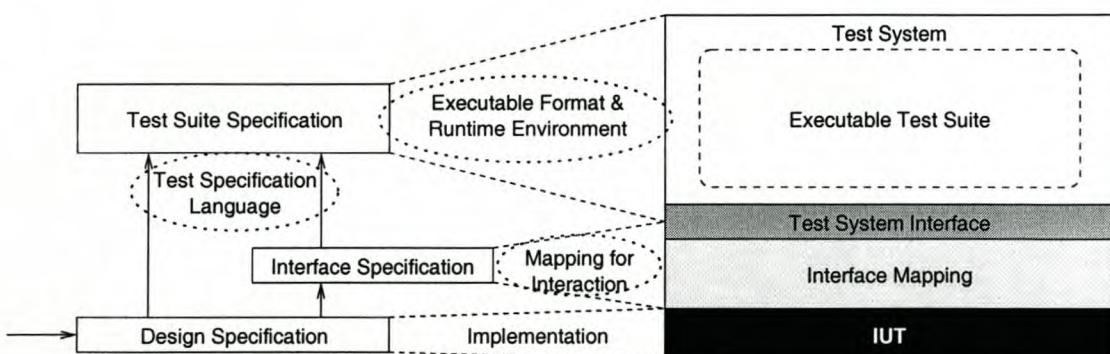


Figure 9: From a design specification to a real test system: The dotted ellipses show the issues considered

To summarise, Figure 9 illustrates the issues considered in this chapter:

- The representation of test cases in a test specification language is discussed in Section 3.1.
- The executable format of a test suite specification and the design of a suitable runtime

environment are discussed in Section 3.2.

- The mapping of abstract interactions in a test suite specification onto real interactions with an implementation under test is discussed in Section 3.3.

### 3.1 Representation of Test Specifications

An executing test case can be seen as a process that emulates a particular scenario of the environment interacting with an implementation under test. A test specification therefore should be able to express the behaviour of such a process in terms of interactions or *communication events*. In addition, a test specification language should support the expression of *alternative event sequences* in cases where more than one possible type of response is expected from an IUT, or even to express possible response failures. Response failures can be determined by using timers and *timeout events* in behavioural expressions. Timing plays an important role in protocols to issue retransmissions, and to decide whether a particular connection is broken. It can also be used to test timing constraints, and to prevent the indefinite waiting of a particular message when an IUT has failed to send it.

Since an implementation under test is a “black box” it is not possible to know whether it is willing to receive or ready to send a particular message. However, a test system should be able to observe every output from an IUT. Hence, a test system must provide the facility to buffer all the messages received from an IUT until they are processed. A notion of *asynchronous communication* is therefore essential in a test specification language.

The need for data types, variable declarations and the specification of message values is invaluable in test case specifications. Usually a PDU (protocol data unit) can be described as a structured data type that should be instantiated with appropriate values before it can be sent to an IUT. When a PDU is received it should match its specification in terms of its field values.

A test case should be able to report a verdict regarding the observable behaviour and whether the purpose of a test case has been achieved. The capability to assign verdicts to specified test sequences will help in the evaluation of an overall test case verdict.

### 3.1.1 CSP as a Test Specification Language

At first it was considered to use a notation like CSP to describe test case behaviour. However, a number of discrepancies with the above-mentioned requirements, which are pointed out in this subsection, do not make CSP suitable as a notation for specifying test cases.

In CSP behaviour is described as communicating processes without shared memory between processes [18]. It makes use of message passing on communication channels between processes. A simple process can be described with an initial event followed by a behaviour described by another process:

$(e \rightarrow P)$  is such a process, which must initially take part in the event  $e$  and then behaves like the process described by  $P$ . This construction is known as *prefixing*.

A process can also behave like one of two processes  $P$  and  $Q$  where the environment ‘decides’ which possibility is enabled:

$(e_p \rightarrow P_p \parallel e_q \rightarrow Q_q)$  is a process where its environment may offer to take part in the events  $e_p$  or  $e_q$ , allowing the process to behave further like  $P_p$  or  $Q_q$  respectively. However, the environment may offer to take part in any of the two, in which case an arbitrary choice is made between them. This construct is known as *external non-determinism*.

In contrast to the above-mentioned requirement of asynchronous communication in test case descriptions, communication between processes in CSP is synchronous, which means that two processes must be willing to engage in the same communication event — one that is willing to send on a channel and another that is willing to receive on that channel. Communication events are described with a “!” and “?” to denote *output* and *input* respectively:

$(c!v \rightarrow P)$  denotes a process which first takes part in a communication event by *outputting* the value  $v$  on channel  $c$ , and then behaves like  $P$ .

$(c?x : M \rightarrow P)$  denotes the process which takes part in a communication event by *inputting* any value of type  $M$  to  $x$  via a channel  $c$ , and then behaves like  $P$ .

It is possible to express asynchronous communication with buffer processes. However, one would expect asynchronous communication semantics underlying a test specification language.

CSP also has constructs for *assignments*, *branching* (if-then-else), and *loops* (while-do). Only an assignment is shown here:

$(x := v)$  is a process where a value  $v$  is assigned to a variable  $x$ .

Processes can be combined as parallel or sequential processes:

$P; Q$  is a process which first behaves like  $P$  and after  $P$  has terminated successfully, it continues to behave like  $Q$ . This is known as *sequential composition*.

$P \parallel Q$  is a process which behaves like  $P$  in parallel with  $Q$ , with synchronized communication over shared channels. This is known as *parallel composition*.

A test case in CSP might look like this:

```

ArpIpTestCase = (utpc0!IPSENDREQUEST → (timer!SET → WaitOnResponse))
WaitOnResponse = (ltpco?arpPacket : {ARPREQUEST} → HandleArpRequest
                  []ltpco?ipPacket : {IPPACKET0} → (timer!CANCEL → PASS)
                  []timer?t : {TIMEOUT} → FAIL)
HandleArpRequest = (ltpco!ARPREPLY →
                     (ltpco?ipPacket : {IPPACKET0} → (timer!CANCEL → PASS)
                     []timer?t : {TIMEOUT} → FAIL))
PASS = (verdict := PASS)
FAIL = (verdict := FAIL)

```

The process *ArpIpTestCase* describes the behaviour of a test case in the scenario sketched on page 16. In the sub-process *WaitOnResponse* it should engage in one of three alternative events, namely to accept either a value *ARPREQUEST* or *IPPACKET0* on channel *ltpco*, or a value *TIMEOUT* on the *timer* channel. Assignment processes, such as *PASS* and *FAIL*, are defined to terminate successfully.

Note that external non-determinism is not appropriate to express alternative test case behaviour. An executing test case should not take part in events that are determined by its environment — it should only *observe* these events. A deterministic evaluation of a list of possible alternative events is necessary to determine what alternative behaviour path should be taken next. Otherwise the behaviour of a test case cannot be predicted and repeated.

In CSP a timer can be regarded as a separate parallel process that communicates to a participating process when a timeout has happened [39]:

```

TIMER = (timer?s : {SET} → (timer?c : {CANCEL} → TIMER
                           []timer!TIMEOUT → TIMER)

```

The process *TIMER* starts a timer when it receives a value *SET* via the channel *timer*, and thereafter it can either accept a value *CANCEL* or send a value *TIMEOUT*.

It is not possible to specify a real time constraint in CSP as to when, for example, a *TIMEOUT* value must be sent. However, explicit timer semantics in a test specification language would be desired, since timers can be considered as part of the environment of a test case.

The behaviour of a closed system is usually described with CSP. In other words every channel is connecting to at least two communicating processes. However, a test specification should only specify the behaviour of test cases in a test system with open-ended channels that should be connected to an unknown implementation under test. The behaviour of an implementation under test can only be stimulated and observed on these connected channels.

### 3.1.2 An Overview of TTCN-3

TTCN-3 is a standardized test specification language. In this subsection only a subset of TTCN-3 is discussed. For a complete language description the reader is referred to the current ETSI ES 201 873-1 standard document [10]. The principle building block of TTCN-3 is a *module*, which can represent a complete test suite or just a library that may be imported by other modules. A module describes a test suite when it contains a control part denoted with the **control** keyword. The execution order of test cases is described in the control part. Program statements such as **if-else** or **do-while** can be used in the control part to specify the selection and possibly repetitious execution of test cases.

```
module MyTestSuite {
  type record IPPacket {...}
  ...
  type component MyMainTestComponent {...}

  template IPPacket IPPacket0 := {...}
  ...
   testcase ArpIpSend() runs on MyMainTestComponent {...}
  ...
  control { //control part
    execute(ArpIpSend());
    ...
  }
}
```

### Test Configurations

The term *test system interface* (TSI) is used in TTCN-3 to specify the points of control and observation of a test system where interaction with a SUT (or test context) takes place. A

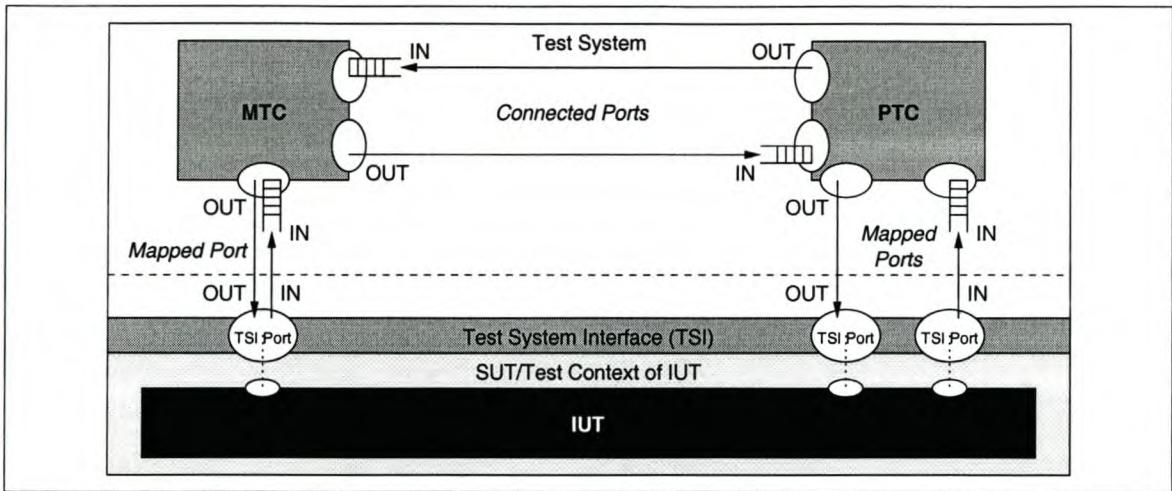


Figure 10: Conceptual view of a TTCN-3 test configuration

test system interface is used to communicate with an IUT via specified *communication ports* in the interface. It is part of a *test configuration*, which can also consist of concurrent test components hosting parallel test case processes. A test system interface and test components are associated with component types. A component type is used to define the ports of a test component or test system interface. Each port is associated with a port type that is declared with lists of allowed message types. Each list is characterised with one of the keywords **in**, **out**, or **inout** indicating message direction on a particular port.

```

type port UpperTesterPortType message {
    out IPSend; //an IPSend typed message can be sent on this port
    in IPDeliver //an IPDeliver message can be received on this port
}
...
type component MyMainTestComponent {
    port UpperTesterPortType UTPort; //to access IUT upper boundary
    port LowerTesterPortType LTPort; //to access IUT lower boundary
    port AuxiliaryPortType AuxPort //just another port
}

type component MyTestSystemType { ... }

```

The conceptual view of a TTCN-3 test configuration in a TTCN-3 test system during the execution of a test case is shown in Figure 10. A communication port is directional and is modelled as an infinite FIFO queue that stores incoming messages until they are processed. This allows the required asynchronous communication. Outgoing messages are directly sent to their destinations. In real implementations queues may overflow, which would result in a runtime error.

The behaviour of a test case is associated with a *main test component* (MTC) that has a type indicated by the keyword **runs on** in a **testcase** declaration. A main test component can also imply the test system interface with its associated ports, except when an explicit component type is associated with the **system** keyword. In the latter case the **system** component type describes the test system interface. *Parallel test components* (PTCs), which are associated with other concurrent behaviours described in **function** declarations, can be created, started and stopped dynamically by the use of **create**, **start**, and **stop** operations respectively. Compatible ports on different test components may be connected to form a communication channel between test components with a **connect** operation. When a test component must communicate through the test system interface with an IUT, a test component port should be explicitly *mapped* to a test system interface port with a **map** operation, unless the test system interface is implied by the test component. It is also possible to disconnect or unmap a previously connected or mapped component port with the **disconnect** or **unmap** operations respectively.

```
function ParallelTestBehaviour() {...}

testcase MyTestCase() //a test case may have a formal parameter list
  runs on MyMainTestComponent system MyTestSystemType //component types of MTC & TSI
{
  var MyPTCType NewComponent; //a reference to a parallel test component

  NewComponent := MyPTCType.create; //create an instance for new component
  connect(NewComponent:Port1, mtc:AuxPort1); //instance of MTC is referenced by mtc
  ...
  map(NewComponent:UTPort, system:UTPCO); //instance of TSI is referenced by system
  map(NewComponent:LTPort, system:UTPCO);
  NewComponent.start(ParallelTestBehaviour()); //start PTC with ParallelTestBehaviour
  //rest of main test case behaviour:
  ...
  NewComponent.done;
  ...
  unmap(NewComponent:UTPort, system:UTPCO);
  disconnect(NewComponent:Port1, mtc:AuxPort1);
}
```

A **testcase** declaration describes the behaviour of a main test component, which is instantiated together with its associated ports when a test case is invoked. In the example above the main test case behaviour is described by test case **MyTestCase**, and runs on the main test component that is of type **MyMainTestComponent**. The test system interface that is associated with this test case is of type **MyTestSystemType**. In addition to the main test component, this test case also dynamically creates a parallel test component of type **MyPTCType**. This test component is referenced by the variable **NewComponent**. The port **Port1** of this test component is connected to the port **AuxPort1** of the main test component, and the ports **UTPort** and **LTPort** are mapped on to the test system interface ports

UTPCO and LTPCO respectively. This test component is then started with the behaviour as specified by function `ParallelTestBehaviour`. After the behaviour of this parallel test component has terminated (indicated with the `done` keyword), the ports should be unmapped or disconnected.

## Data Types and Encodings

TTCN-3 supports a number of basic types that include basic types found in a programming language (such as `integer`, `float`, `char`, and `boolean`), basic string types (such as `bitstring`, `hexstring`, `octetstring`, and `charstring`), structured types (such as `record` and `set`), test configuration types (such as `port` and `component`), and test specific types (such as `verdicttype`). All the types have abstract representations. The latest edition of ASN.1 is also incorporated in TTCN-3. Almost all types, including user-defined types, are valid message types and can therefore be associated with a port type.

```
type integer ubyte (0 .. 255); //integer subtype

var bitstring MyBitstring := '10010111'B;

type octetstring AddressType length(4); //length of AddressType is 4 octets
type record MyRecordType {
    AddressType destinationAddress,
    boolean flag,
    charstring data optional //optional record element
}
```

In this example an integer sub-type with name `ubyte` is declared, which represents integer values in the range of 0 to 255. The variable `MyBitstring` of type `bitstring` consists of 8 bits. The type `AddressType` represents an octet string of 4 octets. The record field `data` in record `MyRecordType` may be omitted in values of this record type as specified with the keyword `optional`.

TTCN-3 allows an association of an encoding rule with a declaration that is used to encode and transfer messages on a communication port, or to decode received messages. This is done with the `encode` and `variant` attributes. Attributes are associated with a so-called `with` statement.

```
type record MyRecordType {
    AddressType destinationAddress,
    boolean flag,
    charstring data optional //optional record element
} with { encode "MyEncodingRule" } //"MyEncodingRule" is defined external to TTCN-3

//unsigned integer type represented on 8 bits:
```

```

type integer uint8 (0 .. 255) with { variant "unsigned 8 bit" };

//signed integer type represented on 32 bits:
type integer int32 (-2147483648 .. 2147483647) with { variant "32 bit" };

```

The record type `MyRecordType` should be encoded by the rule “`MyEncodingRule`” when it is sent via a test system interface port, or decoded by this rule when received via a test system interface port. Similarly the integer types `uint8` and `uint32` should be encoded or decoded as they were represented by a byte and a 32-bit word respectively within the test system.

The specification of the encoding rules is outside the scope of the language. It is the responsibility of the runtime environment to provide appropriate mechanisms to specify and use encoding rules or procedures for encoding and decoding. TTCN-3 does not have a default encoding so that if no specific rule is attributed to a particular declaration an arbitrary encoding is used that is dependent on a particular test system implementation.

### Message Contents and Matching Mechanisms

Descriptions of the contents of messages to be sent and messages that might be received are declared with *message templates*. These descriptions are concerned with the values of messages being sent on a port, and constraints on the message values received. TTCN-3 has a uniform construct for templates of both sending messages and receiving messages. Templates for sending messages are required to specify specific values for all the message fields, while templates for receiving messages may specify general constraints on values with the use of TTCN-3 *matching mechanisms*. Matching mechanisms include specific values, list of values, value ranges, wild-cards for any value, length restrictions, and regular expressions for pattern matching in character strings. Templates are not variable declarations that can be manipulated at runtime, but they may be *parameterized* to allow the specification of more generic message values.

```

//a send template with specific values:
template MyRecordType MyTemplate := {
    destinationAddress := '92E8D485'0;
    flag := true;
    data := "This is the data"
}

//a parameterized receive template:
template MyRecordType MySecondTemplate(charstring dataParam) := {
    destinationAddress := ('92E8D486'0, '92E8D487'0, '92E8D488'0); //list of values
    flag := ?; //any value, i.e., true or false
    data := dataParam //"data" field must match dataParam
}

```

Templates are used in communication operations to create a message value that must be sent, or to match a received message against an expected value specification.

```
AuxPort.send(MyTemplate);

//wait on receive event with message contents specified by MySecondTemplate(?):
AuxPort.receive(MySecondTemplate(?)); //?" as actual parameter means "any value"
```

## Timers

In TTCN-3 timers are declared with the **timer** keyword. A default duration may be assigned to the declaration, but this value can be temporarily overridden with the specification of a duration when a timer is started, i.e., with the **start** operation. A timer may be stopped with the **stop** operation anytime after it has been started. The **timeout** operation is used to wait on a *timeout event* of the specified timer.

```
timer ReceiveTimer := 1.0; //default duration is one second

ReceiveTimer.start(0.3); //start "ReceiveTimer" with a duration of 300ms
...
ReceiveTimer.timeout; //wait on timeout event of "ReceiveTimer"

ReceiveTimer.start; //start "ReceiveTimer" with default duration
...
ReceiveTimer.stop; //stop timer
```

## Verdict Assignment

To automatically evaluate the outcome of a test case, verdicts must be assigned for every test behaviour path. Verdict assignments determine whether an IUT has passed a test case or not. TTCN-3 has a special enumerated type, **verdicttype**, with the values **none**, **pass**, **inconc** (inconclusive), **fail**, and **error**. A verdict value is initialised to **none**, while the **error** verdict may only be set by the test system to indicate that a runtime error has occurred, for example when a port queue overflows. The **setverdict** operation is used to update a verdict value

```
setverdict(pass);
```

TTCN-3 uses overwriting rules when a verdict value is updated. These rules prevent the last verdict update in a test execution sequence from determining the overall verdict of a test case. For example, if the last verdict assignment is **pass**, but a preliminary verdict was **fail**, one would expect the overall verdict to be **fail**. Table 2 lists these rules.

Previous Verdict Value	New Verdict Update Value			
	pass	inconc	fail	none
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail

Table 2: Overwriting rules for a verdict value in TTCN-3

### TTCN-3 Statements

TTCN-3 supports basic programming constructs such as assignments, expressions, conditional **if-else**, and loop constructs (such as **do-while**), which are used for internal control flow and data manipulation. In addition, more test specific behavioural constructs are part of TTCN-3, especially a construct that is similar to the external non-deterministic choice of CSP namely the **alt** statement. Receive and timeout events are used as *guard operations* in an **alt** statement to determine which statement sequence is selected. Each guard operation in turn can also be guarded with a boolean expression between the square brackets ([ ]) in front of each alternative, to enable or disable the evaluation of a particular guard operation. When no boolean expression is present between the square brackets the guard operation is always evaluated. The following example corresponds to the example in Figure 7 that is explained on page 19.

```

testcase ArpIpSend2() runs on ArpIpTestComponent {
    // ArpIpTestComponent also implies test system interface
    var boolean arpReceived := false;
    timer           ReceiveTimer := 1.0;

    UTPCO.send(IpSendRequest);
    ReceiveTimer.start;
    alt {
        [ not arpReceived ] LTPCO.receive(ArpRequest) {
            LTPCO.send(ArpReply);
            arpReceived := true;
            repeat;
        }
        [ ] LTPCO.receive(IpPacket0) {
            ReceiveTimer.stop;
            setverdict(pass);
        }
        [ ] ReceiveTimer.timeout {
            setverdict(fail);
        }
    }
}

```

Before a set of alternatives is evaluated, a “snapshot” is taken of the current state of the components, port queues, and the expired timers. This ensures that the evaluation of a

set of alternatives is an atomic action so that the state cannot be changed while evaluation takes place. Alternatives are evaluated in the order listed until a possible guarding boolean expression and the condition imposed by a guard operation have been fulfilled. When no alternative is enabled a new snapshot is taken and the evaluation is repeated. A re-evaluation of an enclosing alternative construct can explicitly be instructed with a **repeat** operation inside an alternative behaviour.

### 3.1.3 Remarks

The TTCN-3 test specification language is the natural choice when a test specification language has to be selected, because of its status and acceptability as a standard. However, its size and complexity does not make such a choice feasible for the investigation of this thesis. A subset of TTCN-3 was therefore selected and implemented, with a number of syntactical changes that are discussed in the next chapter.

## 3.2 Executable Test Suites and Runtime Support

An *executable test suite* is produced by the translation of a test specification into an executable format. The format would typically be a programming language so that an executable test suite can be *compiled* for execution on a specific test machine, such as the C code generated by the Telelogic Tau tool [44]. Another approach, which is followed with the OpenTTCN Tester [29], is the *interpretation* of test suite specifications. The *TTthree* compiler of Testing Technologies [46] translates TTCN-3 modules to Java code before they are compiled into interpretable Java classes.

The executable format of a test suite should capture the runtime behaviour of a test suite together with its type declarations, template specifications of message values to be sent and matching constraints of message values to be received. The runtime environment of any language normally provides low-level support for abstract language concepts. This could be in the form of a runtime library (runtime system) that should be linked with any generated executable test suite, or a specialized interpreter. The runtime environment of a test system should include support for the handling of communication, timers, as well as the control of actual test case executions.

### 3.2.1 Communication

In TTCN-3 a communication port is modelled by an infinite FIFO queue, because all the output from an implementation under test (or parallel test component) must be observed, and the number of messages is not known beforehand. This, however, is impractical to implement in a test system since a real computer has finite storage. It may therefore be necessary to impose a limit on the number of elements in a communication queue. As suggested in the TTCN-3 standard, a runtime error should be reported for a particular test case when such a queue overflows [10]. A runtime error doesn't mean that a test case has failed; it means that it was impossible to determine a verdict for a particular test case due to a test system limitation or incorrect test case specification.

As mentioned, a test system should provide an interface to map specified communication events onto real communication actions with an IUT. This interface should provide functions to the mapping code that enqueue incoming messages from an IUT on an appropriate port queue. The mapping code should also implement functions in this interface that handle the send events to an IUT as specified by corresponding communication events.

The encoding and decoding of messages can be either part of the test system interface mapping, or the runtime behaviour. With the first approach, the mapping is also concerned with the internal data representations of the test executable, because it should be able to traverse internal message representations to encode a message value into a specified octet-string representation. Conversely, when a message is received from the SUT by the mapping, the decoding involves the construction of an internal value representation. When the mapping should only focus on the transmission of messages to and from a SUT, the second approach is preferable, because a message is encoded before it is sent to the interface mapping, and a message is decoded after it has been received from the interface mapping.

### 3.2.2 Timers

A test case behaviour may depend on the observation of timeout events from running timers in the test system. When all running timers should notify timeouts at one point of observation, all timeout events should be enqueued at that one point. However, this would restrict a test case to observe only one timeout at a time as timeout events are removed from the queue. A test case, for example, may specify alternative behaviours with various timeout guards. Timers therefore should be associated with their own points of observation in a test system so that timeouts can be observed individually.

### 3.2.3 Control and Test Case Execution

The runtime environment should provide mechanisms to start the execution of a test case. An executing test case stimulates a particular implementation under test and observes the resulting behaviour of an implementation. Since each test case tests a specific aspect of an implementation under test, it should not be influenced by any other test case. It is therefore necessary to ensure that only one test case is executed at a time. When test cases with concurrent test components are specified, the underlying concurrency mechanisms of the operating system should be used to implement the required concurrent behaviour. For example, each parallel test component could be implemented as a thread when the whole test system is implemented as one process on some supporting operating system.

The initialization prior to the execution of a test case may include the establishment of test channels (or connections) to the implementation under test (IUT) if a connection-oriented protocol service is used by the IUT. Since a test system interface mapping is concerned with the connection to an IUT, a way to notify this mapping to establish a connection should exist.

#### Snapshot semantics

The guards in alternative behaviour constructs should be evaluated atomically, to make a deterministic choice of which alternative behaviour should be followed. Synchronization mechanisms of an operating system or concurrent programming language should be used, since the states of all the components, message queues, and expired timers that are used during the evaluation may not change. However, this does not mean that new messages and new timeouts may not arrive — they should be buffered in such a way that the evaluation cannot be influenced. One way to overcome this problem is to copy prior to each snapshot evaluation all the relevant data structures that are needed in the evaluation. However, the challenge would be to minimize the number of data structure copies that have to be made, because efficiency plays a role here since an implementation under test cannot be suspended during a snapshot evaluation — it can continuously produce new output that might be evaluated in the further execution of a test case.

#### Test Log

To examine the events after a test execution a mechanism should be provided to log test events as they happen.

### 3.2.4 A TTCN-3 Test System

A TTCN-3 test specification explicitly defines or implies a test system interface in each test case declaration as explained on page 30. However, a test system interface needs to be instantiated and mapped on the interface of a specific test context, which should be connected to an implementation under test. In order to realize such a mapping, the European Telecommunications Standards Institute has specified a standardized runtime interface called the TTCN-3 Runtime Interface (TRI) [13]. The TTCN-3 Runtime Interface is specified in the light of some concepts for a general TTCN-3 test system implementation, which are defined in the same document. These concepts are briefly discussed in this subsection.

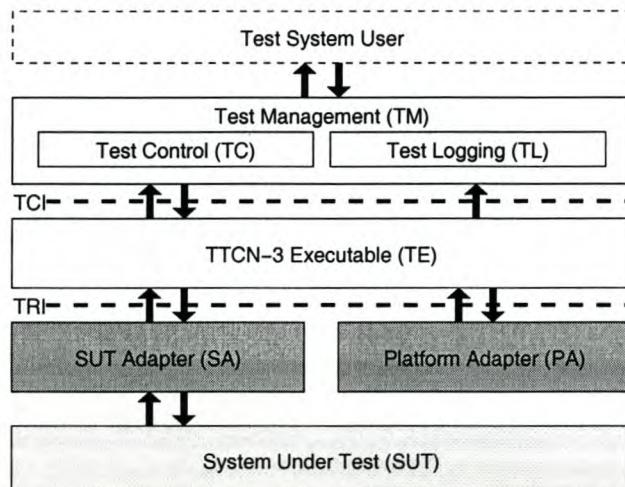


Figure 11: General structure of a TTCN-3 test system

Figure 11 illustrates the general structure of the TTCN-3 test system and environment, consisting of several entities. The TTCN-3 Executable (TE) comprises the interpretation or execution of a TTCN-3 test suite and the necessary runtime functionality. In addition to the already mentioned TRI, another interface is defined on the border of the TTCN-3 Executable, namely the TTCN-3 Control Interface (TCI). The TTCN-3 Control Interface defines the functionality between the TTCN-3 Executable entity and the Test Management entity, which provides a user interface for the control of test execution and logging of test events<sup>1</sup>.

The SUT Adapter (SA) is responsible for the implementation of the mapping between the test system interface and the part of the test context that resides on the test system. Conceptually the SUT Adapter is connected to the SUT, which hosts the target implementation. A

<sup>1</sup>The standardization of the TCI is still in progress.

communication event from the TTCN-3 Executable entity to the SUT is handled by the SUT Adapter entity with a call in the TTCN-3 Runtime Interface (TRI). Conversely, incoming communication messages from the SUT are enqueued in the TTCN-3 Executable entity by the SUT Adapter entity also with a function call that is specified in the TRI. The Platform Adapter (PA) is responsible for the implementation of platform specific functions. These functions include timers, which are dependent on operating system functions, as well as functions that are declared as external to a TTCN-3 module<sup>2</sup>. The functions implemented in the Platform Adapter entity are also accessed via a TRI call.

### TTCN-3 Runtime Interface

The goal of separating the TTCN-3 Executable from other test system entities was to make it easily portable to different platforms and to make it independent of a specific system under test. This means that an executable TTCN-3 test suite relies on abstractions that must be provided on the test machine through a well defined interface. These abstractions include the ports in a specified test system interface (explained on page 29), timers, and external functions. The implementation of these abstractions customizes a particular executable test suite for a specific system under test (in the SUT Adapter) and test platform (in the Platform Adapter). The TTCN-3 Runtime Interface defines a set of standardized operations that should be used in the implementation of these abstractions. The TRI is divided into two sub-interfaces, namely the *TriCommunication* and *TriPlatform* sub-interfaces. The *TriCommunication* sub-interface specifies the interaction between the SUT Adapter and TTCN-3 Executable entities to realize a test system interface mapping and the associated port communication, while the *TriPlatform* sub-interface specifies the interaction between the Platform Adapter and TTCN-3 Executable entities to realize timers and external functions declared in a TTCN-3 module. For illustration purposes, Table 3 lists the correspondence between some of the TTCN-3 operations and the specified TRI operations. Table 4 lists some TRI operations that are used by the adapter entities.

### 3.3 Real Interaction with an IUT

The realization of interaction with a system under test (SUT), and eventually with an implementation under test, involves the mapping of a specified test system interface (TSI) to the test context of an IUT, which includes a protocol service provider and a SUT. The TTCN-3

---

<sup>2</sup>TTCN-3 allows the declaration of external function interfaces.

TTCN-3 Operation Name	TRI Operation Name	TRI Sub-Interface Name
<b>execute</b>	triExecuteTestCase	TriCommunication
<b>map</b>	triMap	TriCommunication
<b>unmap</b>	triUnmap	TriCommunication
<b>send</b>	triSend	TriCommunication
(timer). <b>start</b>	triStartTimer	TriPlatform
(timer). <b>stop</b>	triStopTimer	TriPlatform
<i>TTCN-3 external function</i>	triExternalFunction	TriPlatform

Table 3: Correlation between TTCN-3 operations and TRI operations in the SUT Adapter (via the TriCommunication sub-interface) and in the Platform Adapter (via the TriPlatform sub-interface)

External Events	TRI Operation Name	TRI Sub-Interface Name
<i>Message received from SUT</i>	triEnqueueMsg	TriCommunication
<i>Timer has expired</i>	triTimeout	TriPlatform

Table 4: Correlation between external events observed in the SUT Adapter and Platform Adapter and TRI operations in the TTCN-3 Executable

Runtime Interface (TRI) document calls the entity that implements such a mapping a *SUT Adapter* [13].

For every communication port specified in the test system interface, a test channel should be established between a port instance and an IUT access point in the SUT. Access points are usually available at the lower boundary and upper boundary of protocol entities. An IUT should use an already tested underlying service provider at its lower boundary, since it is assumed that the underlying services used by an IUT are reliable. The same underlying protocol must be used on the test system to establish a test channel to the lower boundary.

Since a protocol entity only provides services to local clients on a machine via its upper boundary, there is usually no support for a test system to access a protocol upper boundary remotely. However, the proposed test architecture for this thesis consists of two machines. One solution is to implement a *test responder* on the SUT, which acts as a user of the IUT, responding to control messages received from the test system via the IUT [32]. This responder includes a test management protocol that uses the IUT. With this approach the test system cannot get reliable control over the upper boundary of the IUT, since the IUT may still have non-conforming behaviour.

The problem can be overcome with the implementation of an “astride test responder”, where one side has access to the IUT and the other side uses a reliable test management protocol to the test system [33]. This also means that a reliable test management protocol must be

implemented on both the test system and the SUT. This solution is illustrated in Figure 12.

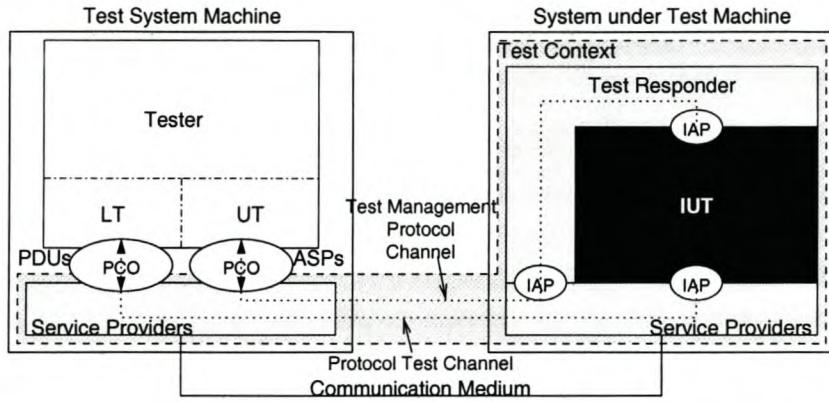


Figure 12: A test architecture consisting of two machines and an Astride Test Responder

The functionality of such a test responder is a mapping between abstract service primitives via the test management protocol and realizations of the service primitives at the upper boundary of the IUT.

### 3.4 Summary

This chapter has focused on some general requirements of a TTCN-3-like protocol test system. A test suite should be specified in a language that is independent of its execution platform and test environment. A runtime environment must supply all the necessary functions to execute a translated test suite and to interface with a mapping to the test context of an implementation under test (IUT). This mapping is called a test adapter, and is used in conjunction with underlying protocol services for real interaction with an IUT. In addition, a test management protocol should be used in conjunction with a test responder to interact with the upper boundary of an IUT.

The next chapter describes the implementation of an experimental protocol test system and a test execution tool for the Native Oberon environment. The implementation is discussed in terms of the issues presented in this chapter.

## Chapter 4

# Implementation of a Tool for Protocol Testing

This chapter describes the implementation of a translation tool and a runtime environment for protocol test specifications. This tool, which is called NOPTT (*Native Oberon Protocol Testing Tool*), is intended to test protocol implementations in the ETH Native Oberon development environment [8]. However, it is still a prototype with many limitations. The specification language accepted by NOPTT is a subset of TTCN-3 with some deviations in data types and syntax. The available data types have a direct mapping to Oberon data types and the syntax was influenced by that of Oberon. There is also no support for concurrent test case behaviour.

A test environment consists of two interconnected machines: one machine hosting a test system (supported by an ETH Native Oberon system) and the other a system under test (SUT), which may be any system that is developed in Oberon. For illustration purposes excerpts from a simple test suite example are used in this chapter. This test suite consists of three test cases in the IP/ARP scenario sketched on page 16. The example test suite is listed in Appendix C.

This chapter starts with an introduction to the task mechanism of the ETH Oberon System. Section 4.2 gives an overview of the test environment. The NOPTT language and some translation aspects are discussed in Section 4.3. Section 4.4 presents the underlying data structures of the runtime environment. It is discussed how these data structures are used by a translated test suite. An example of how a mapping between a test system and a test context can be implemented is described in Section 4.5. Section 4.6 concludes this chapter

with some remarks about the size of the implementation.

## 4.1 The ETH Oberon System

Concurrency is essential in a protocol test system due to the concurrent nature of a protocol environment. The ETH Oberon system is not a true concurrent multitasking system in the sense of preemptive process scheduling. It follows a cooperative non-preemptive scheduling policy [51]. A *central loop* is implemented in which input devices are repeatedly polled for input.

The screen is tiled into rectangular areas called viewers, which may contain text or other visible objects. The general form of a command in ETH Oberon is M.P where P is the name of a procedure and M the name of a module containing the procedure. Any text that is identified by the cursor, i.e., an arrow that follows the movement of the mouse, is interpreted as a command when the middle mouse button is pressed with the cursor on the text.

User input by means of the mouse or keyboard activates an *interactive task* that is handled by a viewer. Mouse clicks are handled by the viewer to which the mouse cursor points. Keyboard input on the other hand is directed at the viewer in focus, i.e., the one in which the *caret* was placed most recently. A *caret* marks the place of subsequent text insertion and is placed at the mouse cursor position with a left mouse click.

While no user input is sensed, user defined *background tasks* are activated periodically. A background task is activated by using a procedure variable that calls a particular *task handler* associated with the task. When a task handler is called, a task is said to be *activated* or *reactivated*. A background task handler is for example used to poll network input and to handle garbage collection. All background tasks are managed by a ring of task descriptors and a pointer to the previously activated task in the ring. This pointer is moved to the next task when the following task is activated. The processor does not make a context switch between tasks and therefore saves valuable processing time.

The central loop and task handlers may only be interrupted by hardware interrupts, when they are allowed, to buffer input data. An Ethernet device for example may buffer incoming Ethernet frames. These frames are later handled by a background task that passes the resulting data to appropriate upper layer protocol handlers.

Tasks should cooperate by returning control to the central loop after a reasonable time interval (less than 100 ms is recommended in [34]). The system cannot prevent an application that

does not cooperate in this regard. However, the user can decide what applications may run and can therefore prevent the execution of tasks that may negatively influence the current need for reactive behaviour. It is important to note that interactive tasks have a higher priority than background tasks.

The above-mentioned properties imply that an ETH Oberon System can be extended to be a dedicated protocol test system without unnecessary background tasks. Since there exists a network polling task, it can also be used (by means of protocol handlers) to enqueue device driver buffered messages at appropriate test system communication ports. The non-preemption property of executing tasks seems to be helpful to the *snapshot semantics* of alternative behaviours (explained on page 35), because while a snapshot is being evaluated by a test case task, port queues cannot be updated. This, however, does not prevent incoming network data from being buffered by a network device driver. When such a device driver buffer overflows, an error should be reported to the test system, because it is assumed that every test interaction would be captured to make an accurate judgement on the behaviour of an implementation under test (IUT). The “openness” of the Oberon system makes it easy to change the sizes of these buffers for specific application needs, for example to reduce buffer overflows to a certain extent.

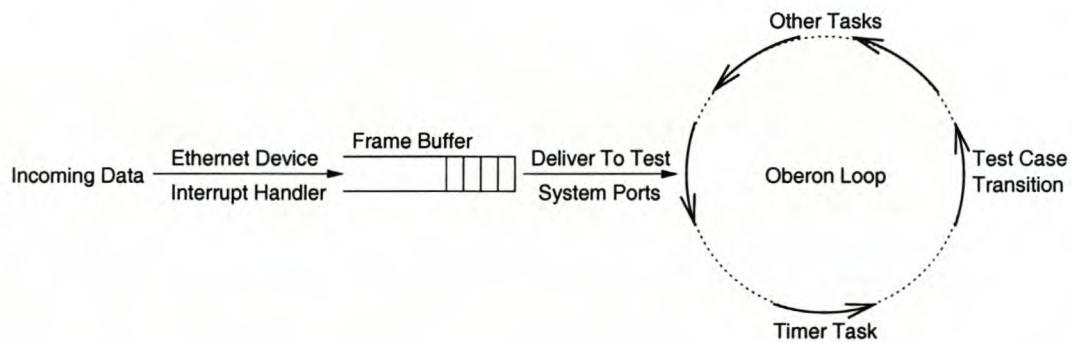


Figure 13: The Oberon Loop and typical tasks in a protocol test system

Figure 13 illustrates the Oberon Loop with typical background tasks in a test system that is produced by using NOPTT. In particular the network polling task is shown that retrieves Ethernet frames from a buffer in the network device driver. These frames are passed to a handler that extracts the messages and enqueues the messages to appropriate communication ports in the runtime environment of the test system.

A test case task in turn can match messages at the ports to the evaluation of specified receive events, and react upon them as specified. Since an Oberon task may be active for a limited time period, and because the progress of a test case is dependent on the processing of other cooperative tasks, it was decided to translate a test case into a state machine. At least one transition is executed per activation, if a transition is enabled in the current state. An Oberon task may also be instructed to be activated at a specified time. It was decided to use this mechanism to implement timers — a timer task is only activated when it must indicate that a timeout has occurred.

Since background tasks have a lower priority than interactive tasks, user interaction should be limited during test execution. This is not considered to be a problem since a skilled user is expected to use the system.

## 4.2 A Test System Overview

A NOPTT test system consists of a translated test suite specification in an executable format, a runtime environment and a necessary mapping of the specified test system interface to a test context of an implementation under test (IUT). This section gives an overview of the modular hierarchy of a NOPTT Test System.

The modular hierarchy is shown in Figure 14. The module that manages the runtime environment is called `PTSRuntime`<sup>1</sup>. It provides functional support during test execution, as well as a communication framework to implement a test system interface mapping. The definition of module `PTSRuntime` is given in Appendix D. Modules `PTSLog` and `PTSQueues` provide an event logging facility, and an abstract data type declaration, `Queue`, representing a FIFO queue, respectively. Every test event is logged by module `PTSLog` in the Oberon Log viewer, which can be inspected during and after test execution. An abstract data type `Port` — for the communication ports in a test system interface — is implemented in module `PTSRuntime` by using module `PTSQueues`.

The aforementioned modules are the fixed modules in a NOPTT Test System. A test specification is translated into two Oberon modules called `PTSTypes` and `PTSETS`. Module `PTSETS`<sup>2</sup> contains all the procedures associated with the behaviour of specified test cases and message templates, while module `PTSTypes` contains the type declarations for the current executable test suite. It was considered that generating two separate modules would be more convenient,

---

<sup>1</sup>The “PTS-” prefix stands for *Protocol Test System*.

<sup>2</sup>The “-ETS” suffix stands for *Executable Test Suite*.

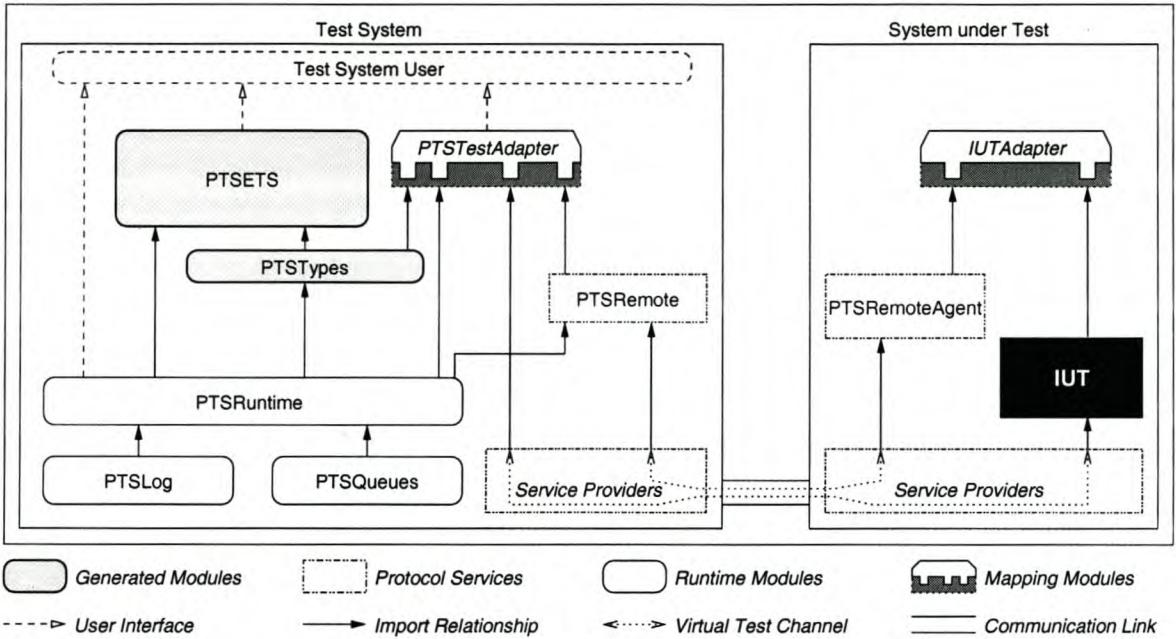


Figure 14: The Modular Hierarchy of a NOPTT Test System

since a generated executable test suite can be large and only the type declarations are needed by the module that implements a test system interface mapping. A test system interface mapping uses the type declarations for the encoding and decoding of messages before transmission and after reception, respectively. The runtime support for these generated modules is discussed in Section 4.4.2.

Module PTSTestAdapter must be implemented by the NOPTT user to realize a mapping of the specified test system interface. This mapping must use appropriate protocol services in the test context to access the IUT. These protocol services usually depend on the type of protocol IUT, and should therefore be implemented on the test machine if they do not exist. Since the interfaces of these protocol services are not known by the runtime environment beforehand, these interfaces should be adapted as expected by the runtime environment. A communication framework was implemented in module PTSRuntime that dictates how PTSTestAdapter should be implemented. This framework is described in Section 4.4.1. Module PTSTestAdapter is sometimes referred to as the *test adapter*.

A special protocol service was implemented in module PTSRemote, which can be used by the test adapter to access the upper service boundary of an IUT remotely. It provides a simple protocol to transfer service primitives between the test adapter and a mapping to the interface of an IUT inside a system under test. On the system under test a module called PTSRemoteAgent must be implemented, which acts as a peer entity for PTSRemote.

The IUTAdapter must be implemented on the SUT to provide a mapping between the service primitives from the remote access protocol and relevant procedure calls in the programming interface of the IUT.

## 4.3 A Test Specification Language and its Translation

The test specification language accepted by the translation tool will be referred to as the *NOPTT language*. The syntax of the language is summarised in Appendix B. Only a simple test specification language was implemented that is basically a subset of TTCN-3. The aim was not to build a fully functional TTCN-3 test system, but to investigate the main issues in the development of a test system tool.

### 4.3.1 A Test Suite

In order to simplify the language, the NOPTT language does not support modules as TTCN-3 does. A test suite is defined in one enclosing **TESTSUITE** block. Since Oberon is the target programming language of an executable test suite, it should be easy to extend the NOPTT language to support modules. The NOPTT language does not have a control part, which means that the control of the execution order and repetition of test cases are left to the user of the test system. As in TTCN-3, no global variables are allowed, because every test case is executed as an isolated entity. In the example below the structure of a test suite in the NOPTT language is shown on the left with a corresponding TTCN-3 module on the right:

<pre> <b>TESTSUITE</b> ARPIPTestSuite;   <b>TYPE</b>     IPAddress = <b>ARRAY</b> 4 <b>OF</b> <b>UINT8</b>;     IPSend = <b>RECORD</b>...<b>END</b>;     ...     <b>TEMPLATE</b> IPSendRequest: IPSend := ...     ...   <b>END</b> IPSendRequest;   ...   <b>INTERFACE</b>   ...   <b>END</b>;   ...   <b>TESTCASE</b> ArpIpSend;   ...   <b>END</b> ArpIpSend;   ... <b>END</b> ARPIPTestSuite. </pre>	<pre> <b>module</b> ARPIPTestSuite {   <b>type</b> integer uint8 (0 .. 255);   <b>type</b> uint8 IPAddress[4];   <b>type</b> <b>record</b> IPSend {...}   ...   <b>template</b> IPSend IPSendRequest := {     ...   }   ...   <b>type</b> <b>component</b> MyMainTestComponent {     ...   }    <b>testcase</b> ArpIpSend()     <b>runs on</b> MyMainTestComponent {       ...     }     ... } </pre>
---	---

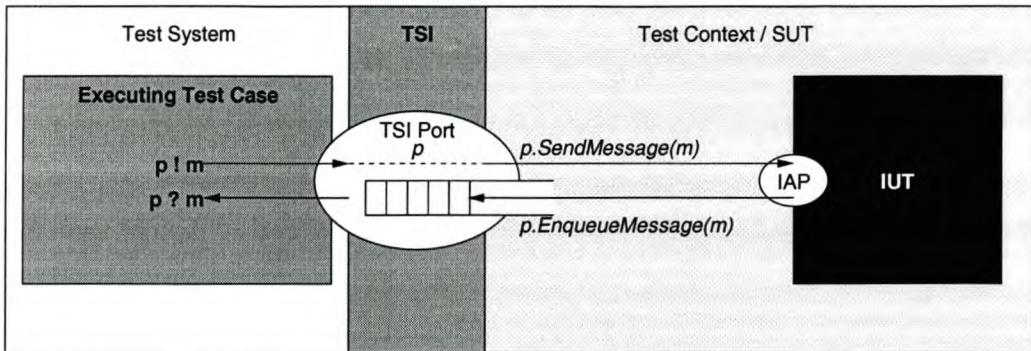


Figure 15: A test configuration in a NOPTT test system

#### 4.3.2 Test System Interface

A test system interface is specified with an **INTERFACE** declaration. In TTCN-3 a test component type must be associated with a test case to define a test system interface that is used by the test case. On the other hand, all test cases in a NOPTT test suite implicitly use the one **INTERFACE** declaration to access the ports in the specified test system interface. Since no concurrent test execution is allowed, no component type declarations are supported. The conceptual view of a test configuration in the NOPTT test system during execution is illustrated in Figure 15. It consists of a fixed number of test system interface ports that are connected with an IUT.

An interface declaration consists of a list of port declarations, each specifying what message types are allowed and in what direction they are communicated on that port. In contrast with TTCN-3 no explicit port type declarations are supported. A port type is declared anonymous as shown in the example below. The example shows the correspondence between an **INTERFACE** declaration in the NOPTT language (on the left) and a component type declaration in TTCN-3 (on the right):

```

INTERFACE
  LTpco: [INOUT ArpPacket, IPPacket];
  UTpco: [OUT IPSend; IN IPDeliver]
END;

type port LowerTesterPortType message {
  inout ArpPacket, IPPacket
}
type port UpperTesterPortType message {
  out IPSend; in IPDeliver
}

type component TestSystemComponentType {
  port LowerTesterPortType LTpco;
  port UpperTesterPortType UTpco
}

```

An interface declaration declares the test system interface used by the current test suite. A connection via this interface to an IUT is established when a test case is started and closed when a test case is terminated.

An interface declaration is translated to an Oberon record type, where each field is a reference to an instance of the Port abstract data type in module PTSRuntime. This data type is explained in Section 4.4.1.

```
TYPE
  INTERFACE = RECORD
    LTpco: RT.Port;
    UTpco: RT.Port
  END;
```

The translator ensures that the communication direction imposed by the port declarations, is maintained in the communication on these ports.

### 4.3.3 Basic Types

Probably the most significant deviation from TTCN-3 can be found in how basic types are handled in the NOPTT language. While a basic numeric type in TTCN-3 does not impose any restrictions on the number of values that can be represented by it, such restrictions are imposed by the NOPTT language. An **integer** variable in TTCN-3, for example, can represent integer values between **-infinity** and **infinity**<sup>3</sup>. A decision was made to closely resemble Oberon types in the NOPTT language, since NOPTT is a tool aimed for the testing of protocols implemented in the Oberon language. It is also assumed the user of this tool is familiar with the representations of the Oberon data types in memory to facilitate the final encoding of messages in a test adapter. The value ranges of types in Oberon are dependent on the target machine of a specific Oberon compiler. For example, an **INTEGER** variable in Oberon represents integer values between, and including, **MIN(INTEGER)** and **MAX(INTEGER)**, which are typically, on a 32-bit processor, between -32768 and 32767<sup>4</sup>. This decision made the mapping of types to Oberon easier during translation because a default representation is associated with each basic type, which is the internal memory representation of the corresponding Oberon type. Note that multi-byte basic types on a 32-bit Intel machine are represented in little-endian byte order.

---

<sup>3</sup>The TTCN-3 keyword **infinity**, which may be used to specify a value range for a numerical sub-type, indicates that there is no lower or upper boundary, depending on the sign in front of it.

<sup>4</sup>An Oberon **INTEGER** is represented by a 16-bit value on 32-bit Intel Architecture.

Simple Type	Oberon Mapping	32-bit Intel Architecture values
BOOLEAN	BOOLEAN	Truth Values <b>TRUE</b> and <b>FALSE</b> (8 bits)
CHAR8	CHAR	Characters of the extended ASCII set (8 bits)
INT8	SHORTINT	8 Bit signed integer values
INT16	INTEGER	16 Bit signed integer values (little-endian)
INT32	LONGINT	32 Bit signed integer values (little-endian)
UINT8	U.UINT8	8 Bit unsigned integer values
UINT16	U.UINT16	16 Bit unsigned integer values (little-endian)
UINT32	U.UINT32	32 Bit unsigned integer values (little-endian)

Table 5: Predefined types of the NOPTT Language

The basic types of the NOPTT language are listed in Table 5 together with the corresponding Oberon types. Due to the specification of non-negative Protocol Data fields, such as lengths, unsigned integer types are also included. Since the Oberon language does not support unsigned integer types, a module UNTS was implemented to support unsigned integers and operations on them. Type conversions between signed and unsigned integers are not supported.

#### 4.3.4 Structured Types and Message Types

The NOPTT language allows the declaration of array and record structures, since these structures are supported by Oberon. In contrast with TTCN-3 where almost any type instance can be used as a message in port communication, only record structures are currently allowed by NOPTT. The reason is that a record type can be translated to a type-extension of a generic record base type for protocol messages. Figure 16 shows the translation of a record type (declaring an IP packet on the left) in the NOPTT language to an Oberon type-extension of the abstract type RT.MessageDesc (on the right). The data type RT.MessageDesc is a generic type that is used to enqueue heterogeneous incoming messages on a port queue. Its declaration is shown in Figure 19 on page 58. Section 4.4.1 describes how this generic type is used.

To specify message fields with bit sizes other than 8-bit, 16-bit, and 32-bit sizes, *bit fields* may be declared inside an unsigned integer record field. These bit fields are similar to those in the C language. This was allowed to declare message fields as close to their transfer representation as possible, and consequently to simplify the final encoding of messages before they are sent.

Bit fields are ordered, in the order of declaration, from the most significant bit to the least significant bit of the indicated unsigned integer field type. The translator ensures that the

<pre> 1  IPAddress = ARRAY 4 OF UINT8; 2 3  IPPacket = RECORD 4 5    version: UINT8:4; 6    IHL: UINT8:4; 7    TOS: UINT8; 8    TotalLen: UINT16; 9    Id: UINT16; 10   Flags: UINT16:3; 11   FragmentOfs: UINT16:13; 12   TTL: UINT8; 13   Protocol: UINT8; 14   HdrChkSum: UINT16; 15   SrcAddr: IPAddress; 16   DstAddr: IPAddress; 17   OptionsOrData: OptdataArray; 18   Data: dataArray 19 END;</pre>	<pre> IPAddress* = ARRAY 4 OF U.UINT8; IPPacket* = POINTER TO IPPacketDesc; IPPacketDesc* = RECORD (RT.MessageDesc)   versionIHL*: U.UINT8;    TOS*: U.UINT8;   TotalLen*: U.UINT16;   Id*: U.UINT16;   FlagsFragmentOfs*: U.UINT16;    TTL*: U.UINT8;   Protocol*: U.UINT8;   HdrChkSum*: U.UINT16;   SrcAddr*: IPAddress;   DstAddr*: IPAddress;   OptionsOrData*: OptArray;   Data*: dataArray END;</pre>
---	--

Figure 16: A NOPTT record type and its translation to Oberon

unsigned integer of the first bit field declaration is filled with successive bit field declarations. The `version` and `IHL` fields shown above are bit fields of size 4 bits each embedded in a single `UINT8` (line 5 in Figure 16). Also, the `Flags` and `FragmentOfs` fields are bit fields that occupy 3 and 13 bits of a `UINT16` field respectively (line 10 in Figure 16). This is how assignments to these fields are translated:

<pre> 1  msg.version := 4; 2 3  msg.IHL := 5; 4 5  msg.Flags := 2; 6</pre>	<pre> msg.versionIHL := S.VAL(U.UINT8, S.VAL(SET, msg.versionIHL)-{4 .. 7} + S.VAL(SET, ASH(S.VAL(U.UINT8, 4), 4))); msg.versionIHL := S.VAL(U.UINT8, S.VAL(SET, msg.versionIHL)-{0 .. 3} + S.VAL(SET, ASH(S.VAL(U.UINT8, 5), 0))); msg.FlagsFragmentOfs := S.VAL(U.UINT16, S.VAL(SET, msg.FlagsFragmentOfs) -{13 .. 15}+S.VAL(SET, ASH(S.VAL(U.UINT16, 2), 13)));</pre>
--	--

The bits of the value that must be assigned are shifted left to the correct position they should have in the allocated unsigned integer, and then bitwise OR-ed with the unsigned integer. The target bits in the allocated unsigned integer are first masked out. The translator ensures that the value can fit in the number of bits available in the bit-field. The “`S.`” qualifier is an alias for the Oberon `SYSTEM` module, which provides low-level functions. The “`SYSTEM.VAL(T, x)`” function interprets *x* as a value of type *T*. `ASH()` is an arithmetic shift function.

Message record fields can be defined in the order in which they should be transmitted. However, because the Oberon compiler aligns any basic type field at an offset that is a multiple of that field’s size, fields have to be declared carefully to ensure that no unexpected “gaps” are produced between fields. Also note that a record is aligned on a 4-byte boundary. Since a 32-bit Intel machine is used, the little-endian byte order of fields in a record (and elements in an array) that have multi-byte basic types, must be changed before a message is sent on

a channel where it is expected to be in a big-endian order. A special function is generated by the translator that changes the byte order of these fields (and array elements) in messages. This function is used in the test interface mapping, where message encodings must be finalized. This is in contrast with TTCN-3 where the final encoding of messages takes place immediately before they are passed to the interface mapping via the TTCN-3 Runtime Interface (TRI) [13].

#### 4.3.5 Message Templates

In contrast to TTCN-3 a distinction is made syntactically between the declaration of sending templates and receiving templates, since there is a semantical difference between these two types of templates: a sending template (denoted with a “:=”) describes the assignment of specific message field values, while a receiving template (denoted with a “=” ) describes a condition, in terms of message fields, that has to be matched by a received message. Templates in the NOPTT language may also be parameterized. The syntax of a template declaration is summarized by the following EBNF:

```

TemplateDeclaration ::= TEMPLATE name [FormalParameters] ":" recordTypeName
                     [TemplateDescription] END .

TemplateDescription ::= ":" AssignmentTemplate | "=" MatchTemplate .

AssignmentTemplate ::= fieldAssignment { ";" fieldAssignment } .

MatchTemplate ::= FieldsBooleanExpression .
  
```

The type associated with a template must be a declared record type. Only the fields of the associated record type and the formal parameters are visible inside the scope of a template. A sending template is described by a list of assignment statements referencing the fields of the associated message type, while a receiving template is described with a boolean expression referencing a message field in each simple predicate, normally separated by a logical AND.

#### Templates for sending messages

A template for sending messages is translated to a function that creates an instance of a particular message type, assigns specific values to message fields, and returns the initialized message instance. Here is an example of a sending template and its translation to Oberon:

```

1  TEMPLATE IpSendRequest2
2  (source, destination: T.IPAddress):
3  IPSSend := 
4
5
6
7  src := source;
8  dst := destination;
9  prot := 0;
10 TOS := 0;
11 TTL := 32;
12 DF := FALSE;
13 Id := 1;
14 BufLen := 1500 - 20;
15 optLen := 0
16
17 END IpSendRequest2;

```

<b>PROCEDURE</b> IpSendRequest2 ( <b>source</b> : T.IPAddress; <b>destination</b> : T.IPAddress): T.IPSSend; <b>VAR</b> msg: T.IPSSend; <b>BEGIN</b> <b>NEW</b> (msg); msg. <b>src</b> := <b>source</b> ; msg. <b>dst</b> := <b>destination</b> ; msg. <b>prot</b> := S.VAL(U.UINT8, 0); msg. <b>TOS</b> := S.VAL(U.UINT8, 0); msg. <b>TTL</b> := S.VAL(U.UINT8, 32); msg. <b>DF</b> := FALSE; msg. <b>Id</b> := S.VAL(U.UINT16, 1); msg. <b>BufLen</b> := S.VAL(U.UINT16, 1480); msg. <b>optLen</b> := S.VAL(U.UINT8, 0); <b>RETURN</b> msg <b>END</b> IpSendRequest2;
---

Sending templates are used in send events — denoted by a “!” — to specify the particular message values that should be sent via a port in the test system interface. The message value described by a sending template may also be assigned to a variable that has the same type. Here is an example:

```

VAR
anIPSSendVariable: IPSSend;
...
BEGIN
...
(* Send a message value described by the "IpSendRequest2" template on the UTpco port: *)
UTpco ! IpSendRequest2(sutIPAdr, testsystemIPAdr);
...
(* Assign the value of type "IPSSend" described by the "IpSendRequest" template to a variable: *)
anIPSSendVariable := IpSendRequest;
...

```

### Templates for receiving messages

A template for receiving messages is translated to a function that receives a reference to a message instance as a parameter and returns whether this instance matches the boolean condition or not. An example together with a translation are shown in Figure 17.

Receiving templates are only used in receive events to specify a matching condition that must be matched by the first message in a port queue to enable the receive event. A receive event — denoted by a “?” — is a guard event that must be used in an alternative statement:

```

ALT
[ ] LTpco ? ArpRequestRcv(sutEthAdr, sutIPAdr, testsystemIPAdr) ->
...
END

```

```

1  TEMPLATE ArpRequestRcv
2  (senderEth: EthAddress;
3   senderIP, targetIP: IPAddress);
4  ArpPacket =
5
6
7  (operation = 1) &
8  (senderHwAddr = senderEth) &
9
10
11
12
13  (senderProtAddr = senderIP) &
14
15
16
17  (targetProtAddr = targetIP)
18
19
20
21
22 END ArpRequestRcv;                                
```

	<b>PROCEDURE</b> MatchArpRequestRcv <b>(</b> senderEth: T.EthAddress; <b>senderIP:</b> T.IPAddress; targetIP: T.IPAddress; <b>msg:</b> T.ArpPacket); BOOLEAN; <b>BEGIN</b> <b>RETURN</b> $\left( \begin{array}{l} \left( U.EQUAL16(msg.operation, S.VAL(U.UINT16, 1)) \& \\ \left( \begin{array}{l} (msg.senderHwAddr[0] = senderEth[0]) \& \\ (msg.senderHwAddr[1] = senderEth[1]) \& \\ (msg.senderHwAddr[2] = senderEth[2]) \& \\ (msg.senderHwAddr[3] = senderEth[3]) \& \\ (msg.senderHwAddr[4] = senderEth[4]) \& \\ (msg.senderHwAddr[5] = senderEth[5]) \end{array} \right) \& \\ \left( \begin{array}{l} (msg.senderProtAddr[0] = senderIP[0]) \& \\ (msg.senderProtAddr[1] = senderIP[1]) \& \\ (msg.senderProtAddr[2] = senderIP[2]) \& \\ (msg.senderProtAddr[3] = senderIP[3]) \end{array} \right) \& \\ \left( \begin{array}{l} (msg.targetProtAddr[0] = targetIP[0]) \& \\ (msg.targetProtAddr[1] = targetIP[1]) \& \\ (msg.targetProtAddr[2] = targetIP[2]) \& \\ (msg.targetProtAddr[3] = targetIP[3]) \end{array} \right) \end{array} \right)$ <b>END</b> MatchArpRequestRcv;
--	---

Figure 17: An example of a receiving template and its translation

#### 4.3.6 Test Case Behaviour

A test case declaration in the NOPTT language is strongly influenced by an Oberon procedure declaration. Timers and variables are declared in a declaration block before a test case body. A test case body is enclosed in a **BEGIN - END** statement sequence block, similar to Oberon. The syntax for communication events in the NOPTT language resemble the syntax of the original TTCN with a “!” and a “?”. A guard event in an alternative statement, i.e., a receive or timeout event, and the following event sequences are separated by an arrow (“->”). An example of a test case is shown in Figure 18. An equivalent test case in TTCN-3 is shown on the right.

The NOPTT language consists of a minimal set of statements that normally do not exist in a programming language. Programming language control constructs such as **IF-ELSIF-ELSE-END** and **WHILE-DO** are not supported. It was reasoned that these constructs can easily be added since they would have a direct mapping to Oberon. The focus was only on special test specification statements, which are communication events, timer events, verdict assignments, and an alternative construct. Assignments are included in NOPTT to have a limited working language. Most of the statements in the NOPTT language are employed in the example of Figure 18:

- A send event is shown on line 7 and 11. In each case a reference to a sending template

<pre> 1  TESTCASE ArplpSend; 2  TIMER ReceiveTimer := 1000; (* ms *) 3  VAR ArpReceived: BOOLEAN; 4  BEGIN 5    ArpReceived := FALSE; 6 7    UTpco ! IpSendRequest; 8    START ReceiveTimer; 9    ALT 10   [ ~ArpReceived ] LTpco ? ArpRequest -&gt; 11     LTpco ! ArpReply; 12     ArpReceived := TRUE; 13     REPEAT 14 15   [ ] LTpco ? IPPacket0 -&gt; 16     CANCEL ReceiveTimer; 17     VERDICT PASS 18 19   [ ] TIMEOUT ReceiveTimer -&gt; 20     VERDICT FAIL 21 22 END 23 END ArplpSend; </pre>	<pre> testcase ArpIpSend() runs on ArpIpTestComponent {   timer      ReceiveTimer := 1.0; // sec    var boolean arpReceived := false;    UTPCO.send(IpSendRequest);   ReceiveTimer.start();   alt {     [ not arpReceived ] LTPCO.receive(ArpRequest) {       LTPCO.send(ArpReply);       arpReceived := true;       repeat;     }     [ ] LTPCO.receive(IPPacket0) {       ReceiveTimer.stop();       setverdict(pass);     }     [ ] ReceiveTimer.timeout {       setverdict(fail);     }   } } </pre>
---	--

Figure 18: The ArplpSend test case in the NOPTT language on the left and its TTCN-3 equivalent on the right

describes the message value to be sent. A variable that has an acceptable message type, may also be used in the place of a sending template.

- In the NOPTT language a receive event may only be used in an **ALT** construct as a guard event as shown on lines 10 and 15. A received message may also optionally be assigned to a variable if it has matched the receiving template. For example, let `msgVar` be a variable of a record type and let `msgTemplate` be a receiving template for that record type, then the following receive event statement assigns a message received on port `LTpco` to variable `msgVar` if this received message has matched the template `msgTemplate`:

```
[ ] LTpco ? msgVar: msgTemplate ->
```

When “`msgVar:`” is omitted, no assignment is taking place. In both cases a receive guard event is only enabled when the first message in the port queue has matched with the receiving template. The message is removed from the queue when this branch of an **ALT** construct was chosen.

- A timer start statement is shown on line 8. A running timer can be cancelled by a timer cancel statement, as shown on line 16. When a timeout has occurred for a specific timer, a timeout guard event in an **ALT** construct is enabled. A timeout guard event is shown on line 19.

- A verdict assignment statement is used on lines 17 and 20.
- An alternative construct, with the keyword **ALT**, is shown from line 9 to line 22. The only repetitive behaviour currently supported is the **REPEAT** statement in an **ALT** construct, as used on line 13. The **REPEAT** statement causes the **ALT** construct to be re-evaluated.
- An assignment is shown on lines 5 and 12.

The translation of a test case to an Oberon task is discussed in Section 4.4.2, after a discussion of the appropriate runtime support.

## 4.4 A Runtime Environment

In this section the communication framework and runtime support are discussed, as implemented in module `PTSRuntime`.

### 4.4.1 A Communication Framework

In object-oriented terminology a *framework* is a set of cooperating classes (or record types in Oberon with procedure variables) that provide a reusable design for a specific class of software [15]. For the NOPTT test system a *communication* framework was implemented using a set of abstract data types in module `PTSRuntime` to dictate the implementation of a test adapter that realizes communication between a test system and an implementation under test (IUT). These data types are explained in this subsection. An example of how this communication framework is used, is described in Section 4.5.

This framework is used to implement a *pluggable* test system interface mapping in the module `PTSTestAdapter`, which can be replaced without the recompilation of the generated test suite modules. This allows the same executable test suite in memory to be used to test different implementations that must conform to the same test specification. It is only necessary to replace module `PTSTestAdapter` when the test context, i.e., the underlying protocol service, has changed. A test system interface mapping may be so general that different test suites can use that same mapping.

The communication framework declares two abstract data types namely `Message` and `Port`. Their declarations are shown in Figure 19. The record types that are declared in a NOPTT

```
Name* = ARRAY MaxNameLen OF CHAR;

(** Message base type *)
Message* = POINTER TO MessageDesc;
MessageDesc* = RECORD (Qs.ItemDesc)
  (** Empty */)
END;

(** Port Type *)
Port* = POINTER TO PortDesc;
SendProcedure* = PROCEDURE (port: Port; msg: Message);
ConnectionProcedure* = PROCEDURE (port: Port);
PortDesc* = RECORD
  name: Name;
  q: Qs.Queue; (* queue for incoming messages */)
  Send: SendProcedure; (* for outgoing messages */)
  Connect, Disconnect: ConnectionProcedure;
  next: Port (* to next port in test system interface port list */)
END;

(** Operations used in a test adapter: */)
PROCEDURE AddPort*(VAR port: Port; name: Name; sendProc: SendProcedure;
  connectProc, disconnectProc: ConnectionProcedure);
PROCEDURE EnqueueMessage*(port: Port; msg: Message; typename: Name);
PROCEDURE Reset*;
(** Runtime Operations used by generated test suite during test execution: */)
PROCEDURE OpenPort*(VAR port: Port; name: Name);
PROCEDURE Send*(port: Port; templatename: Name; msg: Message);
PROCEDURE Receive*(port: Port; templatename: Name; spec: MatchingSpec; VAR msg: Message): BOOLEAN;
```

Figure 19: Type Declarations in the Communication Framework

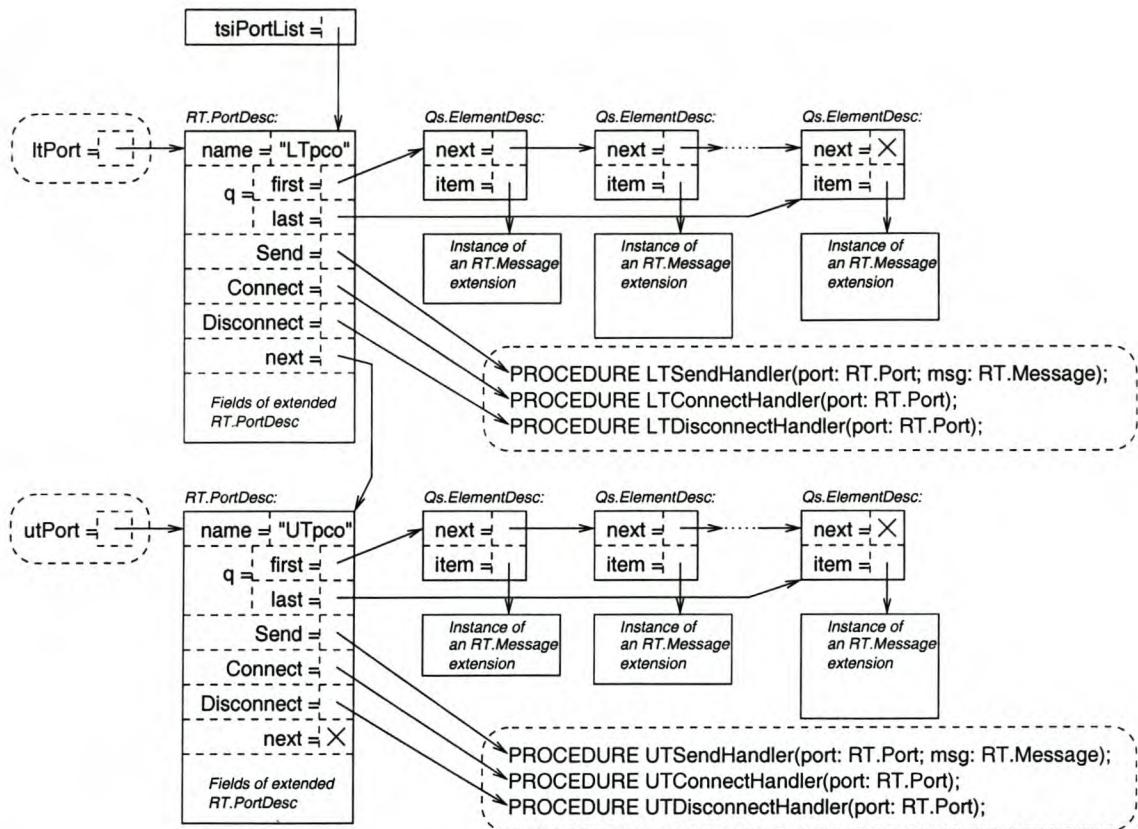


Figure 20: A list of TSI port instances with their message queues in the runtime environment. The dashed-line boxes with rounded corners are procedures and global variables that are part of a test adapter.

test specification are translated to type-extensions of *RT.Message* as shown in Figure 16 on page 52.

A test system interface (TSI) is viewed as a set of communication ports that are connected to an IUT. The conceptual view of a TSI port in the NOPTT test system is shown in Figure 15 on page 49. For each specified TSI port an instance of the abstract type *Port* should be created in the test adapter. An instance that is a type-extension of *Port* can be used to store the state of a connection and to manage a connection via an underlying protocol service to make an IUT accessible for an executing test case. A communication channel may be established by each port instance with an underlying protocol service.

The runtime environment maintains a list of TSI port instances as shown in Figure 20 by variable *tsiPortList* of type *Port*. Procedure *AddPort* is used in a test adapter to add port instances to this list:

Test Suite Event	Procedure Call to Runtime System
<b>START</b> Timer	StartTimer(Timer)
<b>CANCEL</b> Timer	CancelTimer(Timer)
<b>TIMEOUT</b> Timer	Timeout(Timer)
<b>VERDICT PASS</b>	SetVerdict(PASS)
Port ! MsgTemplate(...)	Send(Port, MsgType, MsgTemplate(...))
Port ? MsgVar: MsgTemplate(...)	Receive(Port, MsgType, MsgTemplate(...), MsgVar)

Table 6: Runtime Operations for event statements in the NOPTT language

```

NEW(ItPort);
RT.AddPort(ItPort, "LTpc", LTSendHandler, LTConnectHandler, LTDDisconnectHandler);

```

The global variable `ItPort` is a reference to an instantiated port that is added to the TSI port list in the runtime environment. The parameters `LTConnectHandler` and `LTDDisconnectHandler` are assigned to the `Connect` and `Disconnect` procedure variables of the port instance. They are called before a test case is started and after it has terminated respectively, and can be used to establish and terminate a connection to the SUT. The parameter `LTSendHandler` is assigned to the `Send` procedure variable, which is called to initiate a send event to the IUT on the particular connection associated with the port.

A test adapter uses procedure `EnqueueMessage` to enqueue received messages from the IUT in an appropriate port queue. The formal parameter `typename` of this procedure is used to log that a message of that type is enqueued. The test adapter must ensure that the received message is passed in an instance of an corresponding type-extension of type `Message`, so that the runtime environment can identify its type when it is matched with a receiving template. Procedure `Reset` is called to reset the TSI port list.

The three runtime operations, `OpenPort`, `Send`, and `Receive` are explained in the next subsection.

#### 4.4.2 Runtime Support

The runtime environment provides functional support for the execution of translated test suites. Table 6 lists statements in the NOPTT language in the first column with their corresponding runtime operations in the second column. These operations are discussed in this subsection.

As mentioned, it was decided that test cases and timers are handled with Oberon background

```

(** Test Case Task base type *)
TestCase* = POINTER TO TestCaseDesc;
TestCaseDesc* = RECORD (Oberon.TaskDesc)
    name: Name;
    verdict: SHORTINT;
    running: BOOLEAN
END;

PROCEDURE StartTestCase*(tc: TestCase; name: Name; tchandle: Oberon.Handler);
PROCEDURE StopTestCase*(tc: TestCase);
PROCEDURE SetVerdict*(tc: TestCase; val: SHORTINT);

```

Figure 21: The data type TestCase

tasks. A procedure called a *task handler* is assigned to the *handle* field of an instance of the abstract data type Oberon.Task. Two operations, Oberon.Install and Oberon.Remove, make a task *ready* to be activated or *not ready* respectively. A task is activated when a task handler of a ready task is called by the Oberon Loop.

In module PTSRuntime two abstract data types, TestCase and Timer, are declared. They are type-extensions of Oberon.Task.

### A test case background task

The type TestCase, as shown in Figure 21, stores everything that is needed to manage the execution of a test case in the runtime environment. A test case name is associated with the name field. It is used to log the start and end of a test case execution. The verdict field holds the current verdict of the test case (NONE, PASS, INCONCLUSIVE, or FAIL). NONE is its initial value. The running field indicates whether the test case is running, i.e., whether the test case background task is ready for activation in the Oberon Loop.

A specified test case is associated with a type-extension of the type TestCase as shown in Figure 22 (lines 2 to 8). This extension contains references to the port instances in the test system interface, the current state of the test case state machine, all the local variables of the test case, and references to the declared timers. In this example, type ArplpSendTC is a type-extension of TestCase for the ArplpSend test case. The associated record type ArplpSendDesc contains a field for the test system interface (field *i* on line 4), and a field that stores the current state of the state machine (*state* on line 5). In addition the fields ReceiveTimer and ArpReceived correspond to the locally declared timer and variable in the test case.

A test case is translated to a state machine by using a **CASE** construct. The equivalent state machine of the example test case is shown in Figure 31 and its transition table is

```

1 TESTCASE ArplpSend;
2
3
4
5
6   TIMER ReceiveTimer := 1000; (* ms *)
7   VAR ArpReceived: BOOLEAN;
8
9
10 BEGIN
11
12   ArpReceived := FALSE;
13   UTpco ! IpSendRequest;
14   START ReceiveTimer;
15
16   ALT
17     [ ~ArpReceived ] LTpco ? ArpRequest ->
18
19       ArpReceived := TRUE;
20       LTpco ! ArpReply;
21       REPEAT
22         [ ] LTpco ? IPPacket0 ->
23
24           CANCEL ReceiveTimer;
25           VERDICT PASS
26
27     [ ] TIMEOUT ReceiveTimer ->
28       VERDICT FAIL
29
30   END
31
32 END ArplpSend;
33
34
35
36
37
38
39
40

```

```

TYPE
  ArplpSendTC = POINTER TO ArplpSendTCDesc;
  ArplpSendTCDesc = RECORD (RT.TestCaseDesc)
    i: INTERFACE;
    state: LONGINT;
    ReceiveTimer: RT.Timer;
    ArpReceived: BOOLEAN
  END;
  ...
PROCEDURE ArplpSendHandler(me: Oberon.Task);
BEGIN
  WITH me: ArplpSendTC DO
    CASE me.state OF
      0:
        me.ArpReceived := FALSE;
        RT.Send(me.i.UTpco, "IPSendRequest", IpSendRequest());
        RT.StartTimer(me.ReceiveTimer);
        me.state := 1
      | 1:
        IF ~me.ArpReceived & RT.Receive(me.i.LTpco,
          "ArpRequest", ArpRequest(), dummyMsgVar)
        THEN
          me.ArpReceived := TRUE;
          RT.Send(me.i.LTpco, "ArpReply", ArpReply());
          me.state := 1
        ELSIF RT.Receive(me.i.LTpco, "IPPacket0", IPPacket0(),
          dummyMsgVar)
        THEN
          RT.CancelTimer(me.ReceiveTimer);
          RT.SetVerdict(me, RT.PASS);
          me.state := 2147483647
        ELSIF RT.Timeout(me.ReceiveTimer) THEN
          RT.SetVerdict(me, RT.FAIL);
          me.state := 2147483647
        ELSE
          RT.StopTestCase(me)
        END
      END
    END ArplpSendHandler;

```

Figure 22: The ArplpSend test case on the left and the generated test case descriptor and task handler, in Oberon, on the right

```

(** Timer Task type *)
Timer* = POINTER TO TimerDesc;
TimerDesc* = RECORD (Oberon.TaskDesc)
  name: Name;
  len: LONGINT; (* timer length in milliseconds *)
  running, timeout: BOOLEAN
END;

PROCEDURE NewTimer*(name: Name; duration: LONGINT): Timer;
PROCEDURE StartTimer*(timer: Timer);
PROCEDURE CancelTimer*(timer: Timer);
PROCEDURE Timeout*(timer: Timer): BOOLEAN;

```

Figure 23: The data type Timer

shown in Table 7. The **CASE** construct is shown on the right of Figure 22. Each case in this construct is associated with a state in the state machine, where an enabled transition is selected and executed. With each activation of the test case task, the variable **state** determines the corresponding case that is executed. All the transitions from the current state correspond to different branches in an **IF-ELSIF-END** construct, where each **IF**-condition is an enabling condition for a particular transition. This implies that the first enabled transition encountered is always executed. Since a task handler cannot be preempted, the port queues and expired timers cannot change. A prioritised selection is therefore enforced when alternatives are evaluated that preserves the snapshot semantics of TTCN. When no transition is enabled, the task handler returns to the Oberon Loop without a state change.

### A timer background task

The declaration of type **Timer** is shown in Figure 23. For every timer declared in a test case, an instance of the type **Timer** is created with the operation **NewTimer**. The timer duration in milliseconds is stored in the **len** field. When a timer is started corresponding to a timer start event with the operation **StartTimer**, the **timeout** field is set to **FALSE** and the **running** field is set to **TRUE**. The timer task is put in the Oberon Loop to be activated when it expires. The **timeout** field indicates that a specific timer has expired. The timer task handler is shown in Figure 24.

A timer cancel event causes the operation **CancelTimer** to be called, which removes the timer task from the Oberon Loop if it has not expired yet. A timeout guard event is handled with operation **Timeout**. It returns the value of the **timeout** field for a particular timer.

```

PROCEDURE TimerTaskProc(me: Oberon.Task);
BEGIN
  WITH me:Timer DO Log TIMEOUT event of timer me.name;
    me.running := FALSE; me.timeout := TRUE
  END;
  Oberon.Remove(me) (* remove "me" from the Oberon Loop *)
END TimerTaskProc;

```

Figure 24: The Timer Task Handler

### Starting and terminating a test case

For each test case the translator generates a procedure that acts as an Oberon command, allowing the user to instruct that a test case must be executed. This procedure initializes the local variables and timers of a test case, and binds the test system interface ports to this test case. An example for test case ArplpSend is shown here:

```

PROCEDURE StartArplpSendTC*;
VAR
  TC: ArplpSendTC;
BEGIN
  NEW(TC);
  RT.OpenPort(TC.i.LTpco, "LTpco");
  RT.OpenPort(TC.i.UTpco, "UTpco");
  TC.ReceiveTimer := RT.NewTimer("ReceiveTimer", 1000);
  RT.StartTestCase(TC, "ArplpSend", ArplpSendHandler)
END StartArplpSendTC;

```

Operation `OpenPort` is used to assign a reference to a port in the TSI port list. In addition the procedure variable `Connect` of that port is called. The operation `NewTimer` is explained later. The operation `StartTestCase` makes a test case ready for execution by the Oberon Loop. It ensures that only one test case is ready.

When a test case execution completes, operation `StopTestCase` is called. It removes the test case task from the Oberon Loop and logs the verdict of the finished test case. A call to the procedure variable `Disconnect` is also made.

### Assigning a test case verdict

Operation `SetVerdict` is called every time a **VERDICT** statement is executed in a test case. It updates the current `verdict` variable according to the overwriting rules of TTCN-3, as listed in Table 2 on page 35:

```

PROCEDURE SetVerdict*(tc: TestCase; val: SHORTINT);

```

```

BEGIN
  IF (tc.verdict # FAIL) & (val = FAIL) THEN
    tc.verdict := FAIL
  ELSIF (tc.verdict # INCONCLUSIVE) & (val = INCONCLUSIVE) THEN
    tc.verdict := INCONCLUSIVE
  ELSIF (tc.verdict # PASS) & (val = PASS) THEN
    tc.verdict := PASS
  END
END SetVerdict;

```

### A send event

A send event is handled by the operation Send. On line 16 of Figure 22 the send event is translated to

```
RT.Send(me.i.UTpco, "IPSendRequest", IpSendRequest());
```

This operation calls the procedure variable Send of the port instance referenced by “me.i.UTpco” to send the message value as specified by the template IPSendRequest in the test specification. The generated procedure IPSendRequest() returns this message value. The event is also logged by this operation.

### A receive guard event

A receive guard event is handled by operation Receive. It uses the referenced receiving template to match the first message in the referenced port queue. If that message matches the receiving template, the message is removed from the queue and copied to the formal reference parameter msg. The implementation of this operation is shown in Figure 25.

```

1 PROCEDURE Receive*(port: Port; templatename: Name; spec: MatchingSpec; VAR msg: Message): BOOLEAN;
2   VAR item: Qs.Item;
3   BEGIN
4     item := Qs.FIRST(port.q);
5     IF (item # NIL) & spec.Match(spec, item (Message)) THEN
6       Log "port.name" ? "templatename" event;
7       Qs.DEQUEUE(port.q);
8       msg := item (Message);
9       RETURN TRUE
10    ELSE
11      msg := NIL;
12      RETURN FALSE
13    END
14  END Receive;

```

Figure 25: The runtime operation Receive

```

(** Matching specification base type *)
MatchingSpec* = POINTER TO MatchingSpecDesc;
MatchingSpecDesc* = RECORD
    Match*: PROCEDURE (spec: MatchingSpec; msg: Message): BOOLEAN
END;

```

Figure 26: The data type MatchingSpec

Received messages are matched by using the data type MatchingSpec. Its declaration is shown in Figure 26. For each receiving template a type-extension of this type is generated that contains fields that store the actual parameters of a receiving template. For example, the following type is generated for template ArpRequestRcv that is shown on page 55:

```

ArpRequestRcvMS = POINTER TO ArpRequestRcvMSDesc;
ArpRequestRcvMSDesc = RECORD (RT.MatchingSpecDesc)
    senderEth: T.EthAddress;
    senderIP: T.IPAddress;
    targetIP: T.IPAddress
END;

```

The base type MatchingSpec contains a procedure variable Match that is called by the operation Receive on line 5 in Figure 25 to do the matching. It returns a boolean value indicating if matching was successful or not. With each extension a corresponding matching procedure is assigned that uses the template parameters to do the matching as specified by the template. The matching procedure that is assigned to the type-extension ArpRequestRcvMS above, is as follows:

```

1  PROCEDURE ArpRequestRcvHandler(ms: RT.MatchingSpec; msg : RT.Message): BOOLEAN;
2  BEGIN
3      IF msg IS T.ArpPacket THEN
4          WITH msg: T.ArpPacket DO
5              WITH ms: ArpRequestRcvMS DO
6                  RETURN MatchArpRequestRcv(ms.senderEth, ms.senderIP, ms.targetIP, msg)
7              END
8          END
9      ELSE
10         RETURN FALSE
11     END
12 END ArpRequestRcvHandler;

```

The actual parameters that are stored in the instance of ArpRequestRcvMS are passed to the translated receiving template, procedure MatchArpRequestRcv, as shown on line 6. However, the procedure first does a type test (on line 3) to check that the message to be matched (referenced by parameter msg) is of the expected type. Otherwise the matching has failed due to an unexpected message type in the port queue.

An instance of ArpRequestRcvMS is created and initialized by a generated function procedure that has the name of the associated receiving message template ArpRequestRcv:

```

PROCEDURE ArpRequestRcv(senderEth: T.EthAddress; senderIP: T.IPADress;
    targetIP: T.IPADress): ArpRequestRcvMS;
  VAR
    ms: ArpRequestRcvMS;
BEGIN
  NEW(ms);
  ms.Match := ArpRequestRcvHandler;
  ms.senderEth := senderEth;
  ms.senderIP := senderIP;
  ms.targetIP := targetIP;
  RETURN ms
END ArpRequestRcv;
```

The result of this function procedure is passed to the operation Receive to evaluate this guard event:

```

IF RT.Receive(me.i.LTpco, "ArpRequestRcv",
    ArpRequestRcv(me.sutEthAdr, me.sutIPAdr, me.testsysteIPAdr), dummyMsgVar)
THEN
  ...

```

## 4.5 Interaction with an Implementation under Test

This section describes how interaction between the NOPTT test system and an implementation under test (IUT) is realized using the communication framework. The ARPIPTTestSuite example in Appendix C is used in the discussion. The system under test is a Native Oberon system and the implementation under test is an IP/ARP implementation. The test system is connected to the system under test via an Ethernet cable as shown in Figure 27.

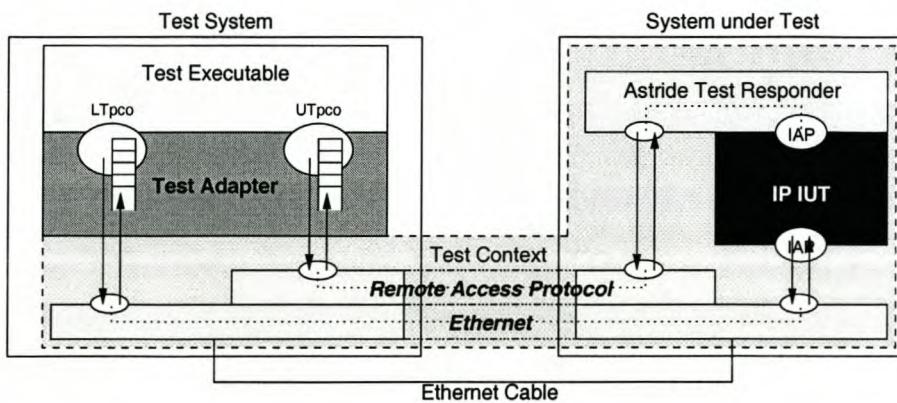


Figure 27: A Test Responder that interacts at the upper boundary of an IUT

#### 4.5.1 The Structure of a Test Adapter

A test developer must implement a module in conjunction with a test specification to realize interaction with an implementation under test. This module is referred to as a test adapter. It involves the instantiation of all the communication ports specified in a test specification by using the communication framework.

Since the implementation under test (IUT) is an IP/ARP implementation and uses Ethernet as a service provider, Ethernet is used to access the lower boundary of the IUT. The port associated with the lower boundary of the IUT should be connected to the Ethernet service on the test system so that messages can be sent and received on that port. Similarly, the port associated with the upper boundary is connected to a special “remote access protocol” service between the test system and system under test (SUT).

The remote access protocol is a simple protocol that uses Ethernet to transfer messages to the upper boundary of the IUT. On the system under test (SUT) the parameters are extracted and the appropriate commands are executed to apply the service primitive to the IUT. Conversely, when a response service primitive from the IUT is observed, its parameters are transferred to the test system by using a specified message type. The adaptations that should be made on the system under test are discussed in Section 4.5.2. Like any message received from the SUT, it is enqueued at the appropriate port instance on the test system.

The listing in Figure 28 shows how the test adapter is initialized for the ARPIPTTestSuite example in Appendix C. The test system interface (TSI) of this example is defined in terms of two ports, “LTpc” and “UTpc”, referring to test channels that have access to the lower boundary and upper boundary of the IUT respectively. A test adapter was implemented in module PTSTestAdapter. It creates two port instances, anchored by the global variables `ltPort` and `utPort`. These port instances are added to the list of TSI ports in the runtime environment with the operation `RT.AddPort`. The module PTSRemote provides the remote access service for the upper boundary. The module that provides an abstraction of network devices in Native Oberon, NetBase, provides an interface where protocol handlers on top of Ethernet can be installed. `NetBase.InstallProtocol` is used to install protocol handlers on top of Ethernet, while `RM.InstallDeliveryHandler` is used to install handlers on top of the remote access protocol.

```

PROCEDURE Init*;
  VAR anIPDeliverInstance: T.IPD Deliver;
BEGIN
  RT.Reset;

  (* — Lower Boundary — *)
  NEW(ltPort);
  RT.AddPort(ltPort, "LTpco", LTSendHandler, LTConnectHandler, LTDDisconnectHandler);

  (* Install handlers for incoming messages from lower boundary via Ethernet *)
  NetBase.InstallProtocol(ArpPacketReceived, idARP);
  NetBase.InstallProtocol(IPPacketReceived, idIP);
  ...

  (* — Upper Boundary — *)
  NEW(utPort);
  RT.AddPort(utPort, "UTpco", UTSendHandler, UTConnectHandler, UTDisconnectHandler);

  (* Install handlers for incoming messages from upper boundary via remote access protocol *)
  NEW(anIPDeliverInstance);
  RM.InstallDeliveryHandler(IPDeliverReceived, T.idIPDeliver, anIPDeliverInstance, SIZE(T.IPD DeliverDesc));
  ...
END Init;

```

Figure 28: The initialization procedure for the PTSTestAdapter module

### Access to lower boundary via Ethernet

The customised procedures which have to be associated with ltPort are LTSendHandler, LTConnectHandler, and LTDDisconnectHandler. The latter two procedures are called immediately before a test case execution is started (to establish connections via a service provider to the SUT) and immediately after a test case has terminated (to close connections) respectively. In this example, however, Ethernet is not connection-oriented and therefore no connections have to be established. However, when a connection-oriented service is used, an extended port type can be used to hold the variables associated with a connection.

The LTSendHandler is called when a send operation on the “LTpco” port is executed. Its structure is shown in Figure 29. A NetBase.Item is constructed consisting of the encoded message to be sent. The message types supported, i.e., T.ArpPacket (line 7) and T.IPPacket (line 13), are those that are associated with “LTpco” in the test system interface declaration (on page 49). Only a change of byte order is necessary before the messages are ready to be sent (on lines 11 and 17).

One of the limitations of the NOPTT language is that it cannot specify calculations on an encoded message value. The IP checksum therefore has to be calculated after the byte order of the numerical fields have changed (line 18).

```

1 PROCEDURE *LTSendHandler(port: RT.Port; msg: RT.Message);
2   VAR
3     item: NetBase.Item; protocolNo: LONGINT; len: INTEGER;
4   BEGIN
5     NetBase.NewItem(item);
6
7     IF msg IS T.ArpPacket THEN
8       WITH msg: T.ArpPacket DO
9         protocolNo := IdARP;
10        len := SIZE(T.ArpPacketDesc);
11        T.ByteOrder(msg) (* Byte order is changed to network byte order *)
12      END
13    ELSIF msg IS T.IPPacket THEN
14      WITH msg: T.IPPacket DO
15        protocolNo := IdIP;
16        len := msg.TotalLen;
17        T.ByteOrder(msg); (* Byte order is changed to network byte order *)
18        msg.HdrChkSum := IPChecksum(msg↑)
19      END
20    ELSE
21      error := TRUE; (* unknown message to be sent over Ethernet *)
22    RETURN
23  END;
24  Copy len bytes of msg contents to NetBase item
25  ...
26  port.ethernet.Send(port.ethernet, SHORT(protocolNo), port.etherDest, item)
27 END LTSendHandler;

```

Figure 29: The structure of the LTSendHandler

The ARPPacketReceived and IPPacketReceived procedures (referenced in the listing shown in Figure 28) are invoked when an ARP or IP packet are processed respectively by the NetBase polling task. The IPPacketReceived procedure is listed in Figure 30. The contents of the received NetBase.Item, which is an IP packet, is then copied into an T.IPPacket instance. Numerical fields are changed to the expected byte order of the host machine, before the message is enqueued on the port queue associated with the lower boundary of the IUT, namely ltPort. The ArpPacketReceived procedure has the same structure.

```

PROCEDURE *IPPacketReceived(item: NetBase.Item);
  VAR ipPacket: T.IPPacket; (* Message type generated in PTSTypes module *)
BEGIN
  NEW(ipPacket);
  Copy data contents received in item to ipPacket
  ...
  T.ByteOrder(ipPacket); (* Byte order is changed to host byte order *)
  RT.EnqueueMessage(ltPort, ipPacket)
  ...
END IPPacketReceived;

```

Figure 30: The IPPacketReceived procedure that is called to process IP packets

### Access to upper boundary via a *remote access protocol*

A remote access protocol is implemented in module PTSRemote and is used by the customised procedures associated with the utPort instance. The IPDeliver primitive from the upper boundary of the IP implementation is handled with an up-call from the remote access protocol to the IPDeliverReceived procedure:

```
PROCEDURE *IPDeliverReceived(msg: RT.Message): RT.Message;
BEGIN
  WITH msg: T.IPPacket DO
    RT.EnqueueMessage(utPort, msg);
    NEW(msg)
  END;
  RETURN msg
END IPDeliverReceived;
```

The UTSendHandler passes a message received from the runtime module to the remote access protocol. In the example only the IPSend primitive is supported:

```
PROCEDURE *UTSendHandler(msg: RT.Message);
BEGIN
  IF msg IS T.IPSend THEN
    RM.Send(msg (T.IPSend) ↑, T.idIPSend, SIZE(T.IPSendDesc))
  END
END UTSendHandler;
```

#### 4.5.2 An Adaptation in the System under Test

A developer of the implementation under test must provide the necessary adaptations in the system under test to realize interaction between the remote access protocol and the upper boundary of the implementation under test. The underlying protocol service of an implementation under test is usually sufficient to access its lower boundary. However, to access the upper boundary of an implementation under test (IUT) a mapping between service primitives and procedure calls in the programming interface of the IUT is necessary. This also includes the initialization of actual parameters for these procedure calls. For example, the following procedure on the system under test maps an IPSend message from the test system to a NetIP.IPSend call in the Native Oberon system:

```
1 PROCEDURE * IPSendPrimitive(msg: PTS.Message): PTS.Message;
2   VAR item: NetBase.Item;
3   BEGIN
4     WITH msg: IPSend DO
5       NetBase.NewItem(item);
6       DEC(item.ofs, msg.BufLen);
7       INC(item.len, msg.BufLen);
8       S.MOVE(S.ADR(msg.Buffer[0]), S.ADR(item.data[item.ofs]), msg.BufLen);
```

```

9
10    NetIP.IPSend(msg.src, msg.dst, msg.prot, item)
11  END;
12  RETURN msg
13 END IPSendPrimitive;

```

The data that are sent by IP are copied into an appropriate buffer (on lines 5 to 8). On line 10 the call to IP is made to send an IP packet.

## 4.6 Remarks

A functional subset of TTCN-3 was used as an experimental test specification language. A translator for the experimental language was implemented that generates Oberon code. The translator builds a syntax tree and a symbol table, from which the Oberon code is generated. The translator comprises approximately 4000 lines of code, which is by far the largest component of the testing tool. The code of the runtime module is about 500 lines.

The example IP/ARP abstract test suite consists of approximately 260 lines, which are translated to about 700 lines of Oberon code. This shows that a test specification language is indeed a tool that simplifies test cases development, without the programming detail of an executable test case.

A test adapter must be implemented to realize specified interactions with an IUT. In addition, some extra code must be provided in the SUT to realise interactions with the upper boundary of a protocol implementation. This code is supported by the implementation of a simple remote interface access protocol of about 140 lines in both the test system and SUT. The total amount of extra code on the test system and the SUT for the IP/ARP example, for which only two test system ports were defined, consists of roughly 270 lines.

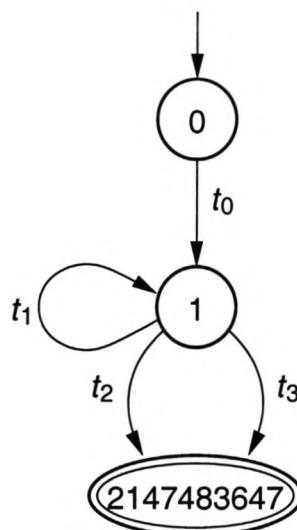


Figure 31: A Finite state machine for the ArplpSend test case

Transition	State	Condition	Action (Possibly guarded by first statement)	Next State
$t_0$	0	TRUE	ArpReceived := FALSE; UTpco!IpSendRequest; <b>START</b> ReceiveTimer	1
$t_1$	1	$\sim$ ArpReceived	LTpco?ArpRequest -> ArpReceived := TRUE; LTpco!ArpReply	1
$t_2$	1	TRUE	LTpco?IPPacket0 -> <b>CANCEL</b> ReceiveTimer; <b>VERDICT PASS</b>	2147483647
$t_3$	1	TRUE	<b>TIMEOUT</b> ReceiveTimer -> <b>VERDICT FAIL</b>	2147483647

Table 7: Transition table for the ArplpSend test case

## Chapter 5

# Conclusion

This thesis investigated the development of a tool that helps in the automation of specified test execution. In retrospect implementing a test system was not a difficult task. However, a number of important issues are summarized here because they might be of value when new test systems have to be implemented.

A protocol implementation must conform to a particular specification (i.e., a protocol standard) to ensure that it will behave as expected in its operational environment. *Conformance testing*, which was the approach followed in this thesis, is an approach that focuses on the observable behaviour of a “black box” implementation without any knowledge of its implementation.

Since a conformance test system is not concerned with the code of an implementation under test, the internal behaviour of an implementation under test is not monitored. An implementation under test also cannot be suspended by a test system as an interactive debugger can. Hence, a test system must always be able to observe the output from an implementation under test. An *asynchronous communication* model is therefore essential in a conformance testing system. Output messages from the different access points at an implementation under test must be enqueued at corresponding observation points in the interface of a test system from where an executing test case can match the expected messages. A failure in the observable behaviour of an implementation under test can be detected in this way. Normally a failure is revealed when an unspecified message is encountered or if an expected message is absent after a specified time interval.

It is important to retain the order in which output is produced, otherwise it is not possible to evaluate the behaviour of an implementation under test. A test specification should specify

all the acceptable orderings in which output messages may arrive at the test system. It should therefore be possible to specify alternative behaviours of test behaviour in one test case. An executing test case, for example, can wait on the occurrence of any output from a number of alternatives.

Timers should also be part of a test environment so that timeouts can be observed by an executing test case. They are observable event sources as are the observation points in the interface of a test system. Timeout events therefore should not be enqueued on one queue so that only one timeout is observable at a given time.

Test programs can be written from scratch in a programming language or they can be derived from test specifications. In the latter case, when a test specification language is used, test programs can be automatically generated. The advantage of this approach is that a test developer need not be concerned with how to implement a test system. A test specification language provides abstractions of implementation details.

When a test system is realized, the interface of a particular test context should be adapted to the specified interface of a test system to enable interaction between the test system and a specific implementation under test. During test execution a test context facilitates communication channels to the implementation under test. It is desirable that these connections are reliable, but it may not always be possible, for example when an underlying protocol service of an implementation under test does not guarantee delivery. It should then be considered to emulate an underlying protocol service with a reliable protocol in both the system under test and the test system.

To support a test system, an operating system should provide certain functions. These functions are concurrency, a timer service, and network protocol services. An operating system must at least provide limited support for concurrency, because a protocol environment has a concurrent nature and test cases might consist of several concurrent components. Synchronization mechanisms should be available to preserve the atomic evaluation of alternative guards.

An implementation under test usually makes use of underlying protocol services, through which a test system would access its lower boundary. These protocol services therefore should be available on the test machine. These protocols should be reliable to ensure that the implementation under test will receive the specified messages and that output messages from the implementation under test are observed correctly on the test system. In addition, when the service boundary of an implementation under test is also needed to be accessible by a test

system, a separate reliable protocol service may be needed for this purpose.

There are two properties of the ETH Oberon system architecture that simplify the implementation of test systems for protocols. These properties are its simple task mechanism and its “openness” in terms of access to device drivers. The non-preemption property of the ETH Oberon tasking mechanism makes it easy to preserve the snapshot semantics during the evaluation of alternative guards. Since timeouts and incoming communication messages are reported and enqueued respectively by cooperative tasks, the states of timeouts and queues cannot be changed when a background task that executes a test case is active. By ensuring that a complete snapshot evaluation is done by one task activation, it is not necessary to make copies of the snapshot state before it is evaluated.

The ETH Oberon system executes in a single address space, which makes accessing device drivers easy and efficient during testing. The size of the system and its clear modular interfaces make it relatively easy to understand the system to add device drivers. This makes the ETH Oberon system attractive for the testing of lower layer protocols.

## 5.1 Future Work

A bigger subset of TTCN-3 would be needed for realistic test specifications. Currently only a small subset is supported. To allow data types that have abstract representations, such as the types in TTCN-3 and ASN.1, a framework of abstract data types could be implemented in Oberon. Each abstract data type should correspond to a basic type supported in the test specification language so that a value of a basic type can have an abstract representation in the test system. In addition an abstract data type should be implemented for each class of structured types that is supported. It must be possible to inspect message values in these abstract representations to encode them for transmission on an underlying protocol service, and to decode received messages into these abstract representations.

## Appendix A

### List of Abbreviations

**ASN.1** Abstract Syntax Notation One

**CSP** Communicating Sequential Processes

**CTMF** Conformance Testing Methodology and Framework

**ETSI** European Telecommunications Standards Institute

**FSM** Finite State Machine

**IAP** Implementation Access Point

**ISO** International Standards Organisation

**IUT** Implementation Under Test

**LOTOS** Language of Temporal Ordering Specifications

**MSC** Message Sequence Chart

**MTC** Main Test Component

**OSI** Open Systems Interconnection

**PA** Platform Adapter

**PCO** Point of Control and Observation

**PDU** Protocol Data Unit

**Promela** Process Meta Language

*APPENDIX A. LIST OF ABBREVIATIONS*

78

**PTC** Parallel Test Component

**SDL** Specification and Description Language

**SA** SUT Adapter

**SP** Service Primitive

**SUT** System Under Test

**TE** TTCN-3 Executable

**TRI** TTCN-3 Runtime Interface

**TSI** Test System Interface

**TTCN** Tree and Tabular Combined Notation

**TTCN-3** Testing and Test Control Notation (Version 3)

## Appendix B

# Syntax of Test Specification Language

The syntax of the experimental test specification language is given in the following EBNF.  $[\alpha]$  denotes the sentence  $\alpha$  or the empty sentence and  $\{\alpha\}$  denotes a finite sequence of sentences  $\alpha$  or the empty sentence. Keywords are all bold capital letter words and are not put between quotes in the EBNF. Other terminal symbols are enclosed in double quotes.

### B.1 Declarations

```
TestSuite ::= TESTSUITE name ";" TSDeclarationSequence END name "." .  
TSDeclarationSequence ::= {TYPE {TypeDeclaration ";" } | {TemplateDeclaration ";" }}  
    InterfaceDeclaration ";" {TestcaseDeclaration ";" } .  
TypeDeclaration ::= name "=" Type .  
Type ::= name | StructuredType .  
StructuredType ::= ArrayType | RecordType .  
ArrayType ::= ARRAY length OF Type .  
length ::= ConstantExpression .  
ConstantExpression ::= expression .  
RecordType ::= RECORD FieldList {";" FieldList} END .  
FieldList ::= [NameList ":" FieldType] .  
NameList ::= name {"," name} .  
FieldType ::= name [":" numOfBits] | StructuredType .
```

```

numOfBits ::= number .

TemplateDeclaration ::= TEMPLATE name [FormalParameters] ":" recordTypeName
    [TemplateDescription] END .

recordTypeName ::= name .

FormalParameters ::= "(" [FPSection {";" FPSection}] ")" .

FPSection ::= name {"," name} ":" FormalType .

FormalType ::= name .

TemplateDescription ::= "==" AssignmentTemplate | "=" MatchTemplate .

AssignmentTemplate ::= fieldAssignment {";" fieldAssignment} .

fieldAssignment ::= assignment .

MatchTemplate ::= FieldsBooleanExpression .

FieldsBooleanExpression ::= expression .

InterfaceDeclaration ::= INTERFACE PortDeclaration {";" PortDeclaration} END .

PortDeclaration ::= [NameList ":" "[" TypeListSequence "]"] .

TypeListSequence ::= TypeList {";" TypeList} .

TypeList ::= [( OUT | IN | INOUT ) NameList] .

TestcaseDeclaration ::= TESTCASE name ;"
    [TIMER {TimerDeclaration ";" }] [VAR {VariableDeclaration ";" }]
    [BEGIN EventSequence] END name .

TimerDeclaration ::= NameList "==" ConstantExpression .

VariableDeclaration ::= NameList ":" Type .

```

## B.2 Event Statements

```

EventSequence ::= NormalEvent {";" NormalEvent} .

NormalEvent ::= [ assignment | SendEvent | AlternativeEvents | StartTimerEvent |
    CancelTimerEvent | VerdictUpdate | REPEAT ] .

assignment ::= designator "==" expression .

designator ::= name {"." name} | "[" expression "]" .

SendEvent ::= PortName "!" designator [ActualParameters] .

PortName ::= name .

MessageTypeName ::= name .

ActualParameters ::= "(" [ExpList] ")" .

ExpList ::= expression "," expression .

```

AlternativeEvents ::= **ALT** AlternativeList **END** .  
 AlternativeList ::= { "[" [ BooleanExpression ] "]" GuardEvent "→" EventSequence} .  
 BooleanExpression ::= expression .  
 GuardEvent ::= TimeoutEvent | ReceiveEvent .  
 TimeoutEvent ::= **TIMOUT** TimerName .  
 TimerName ::= name .  
 ReceiveEvent ::= PortName "?" [designator ":" ] TemplateName [ActualParameters] .  
 TemplateName ::= name .  
 StartTimerEvent ::= **START** TimerName .  
 CancelTimerEvent ::= **CANCEL** TimerName .  
 VerdictUpdate ::= **VERDICT** VerdictValue .  
 VerdictValue ::= **PASS** | **FAIL** | **INCONCLUSIVE** .

### B.3 Expressions

expression ::= SimpleExpression [relation SimpleExpression] .  
 relation ::= "=" | "#" | "<" | "<=" | ">" | ">=" .  
 SimpleExpression ::= ["+"|"-"] term {AddOperator term} .  
 AddOperator ::= "+" | "-" | **OR** .  
 term ::= factor {MulOperator factor} .  
 MulOperator ::= "\*" | "/" | **DIV** | **MOD** | "&" .  
 factor ::= "(" [expression {"," expression}] ")" | set | "¬" factor | number | CharConstant |  
**TRUE** | **FALSE** | string | designator [ActualParameters] .  
 set ::= "{" [SetElement {"," SetElement}] "}" .  
 SetElement ::= expression [.." expression] .

### B.4 Tokens

name ::= letter {letter | digit} .  
 letter ::= "a" | ... | "z" | "A" | ... | "Z".  
 digit ::= "0" | ... | "9".  
 number ::= integer .  
 integer ::= digit {digit} | digit {hexDigit} "H" .

```
hexDigit ::= digit | "A" | "B" | "C" | "D" | "E" | "F" .  
CharConstant ::= "" character "" | digit {hexDigit} "X" .  
string ::= "" {character} "" .  
character ::= Any printable ASCII character.
```

## B.5 Comments

Comments are delimited by “(\*” and “\*)” and may be nested.

## B.6 Predefined Type Names

BOOLEAN	CHAR8
INT8	UINT8
INT16	UINT16
INT32	UINT32

## Appendix C

# Test Suite Example

A complete test suite example is given here in the syntax of the experimental test suite language developed in this thesis. The implementation under test (IUT) is a Native Oberon Internet Protocol (IP) implementation and the Ethernet Address Resolution Protocol (ARP).

```

1 TESTSUITE ARPIPTestSuite; (* IPv4 *)
2
3 (* ----- TYPES ----- *)
4 TYPE
5   EthAddress = ARRAY 6 OF UINT8;
6   IPAddress = ARRAY 4 OF UINT8;
7
8   IPSSend = RECORD (* ASP TO UT *)
9     src: IPAddress; (* source address *)
10    dst: IPAddress; (* destination address *)
11    prot: UINT8; (* protocol *)
12    TOS: UINT8; (* type of service *)
13    TTL: UINT8; (* time to live *)
14    DF: BOOLEAN; (* don't fragment *)
15    Id: UINT16; (* identifier *)
16    BufLen: UINT16; (* data length *)
17    Buffer: ARRAY 1500 OF UINT8; (* data (user PDU) *)
18    opt: ARRAY 40 OF UINT8; (* option data *)
19    optLen: UINT8; (* option data length *)
20  END;
21
22  IPDeliver = RECORD (* ASP FROM UT *)
23    src: IPAddress; (* source address *)
24    dst: IPAddress; (* destination address *)
25    prot: UINT8; (* protocol *)
26    TOS: UINT8; (* type of service *)
27    BufLen: UINT16; (* data length *)
28    Buffer: ARRAY 1500 OF UINT8; (* data (user PDU) *)
29    opt: ARRAY 40 OF UINT8; (* option data *)
30    optLen: UINT8; (* option data length *)
31  END;
32
33  ArpPacket = RECORD (* for ip-over-ethernet *) (* PDU TO, FROM LT *)
34    hwType: UINT16; (* hardware type (= 1) *)
35    protType: UINT16; (* protocol type (= 800H) *)
36    hwLen: UINT8; (* hardware length (= 6) *)

```

## APPENDIX C. TEST SUITE EXAMPLE

```

37     protLen: UINT8; (* protocol length (= 4) *)
38     operation: UINT16; (* operation = 1 for request or operation = 2 for reply *)
39     senderHwAdr: EthAddress; (* sender hardware address *)
40     senderProtAdr: IPAddress; (* sender protocol address *)
41     targetHwAdr: EthAddress; (* target hardware address *)
42     targetProtAdr: IPAddress; (* target protocol address *)
43   END;
44
45   IPPacket = RECORD (* PDU TO, FROM LT *)
46     version: UINT8:4; (* Version *)
47     IHL: UINT8:4; (* Internet Header Length: unit of 32bit words *)
48     TOS: UINT8; (* Type of Service *)
49     TotalLen: UINT16; (* Total Length of datagram *)
50     Id: UINT16; (* Identification *)
51     Flags: UINT16:3; (* Control flags: [0 ; Don't Fragment ; More Fragments] *)
52     FragmentOfs: UINT16:13; (* Fragment Offset: units of 8 octets (64 bits) *)
53     TTL: UINT8; (* Time to Live *)
54     Protocol: UINT8; (* Protocol of next level used in data portion *)
55     HdrChkSum: UINT16; (* Header Checksum *)
56     SrcAdr: IPAddress; (* ARRAY 4 OF UINT8 *)
57     DstAdr: IPAddress; (* ARRAY 4 OF UINT8 *)
58     OptionsOrData: ARRAY 40 OF UINT8;
59     Data: ARRAY 1440 OF UINT8
60   END;
61
62   (* ----- TEMPLATES ----- *)
63
64   TEMPLATE IpSendRequest: IPSend :=
65     src := (146, 232, 212, 203); (* SUTIP *)
66     dst := (146, 232, 212, 134); (* TSIP *)
67     prot := 0;
68     TOS := 0;
69     TTL := 32;
70     DF := FALSE;
71     Id := 1;
72     BufLen := 1500 - 20;
73     (* Buffer[i] := CHR(ORD("0") + i MOD 10); *)
74     (* opt[0] := 0; *)
75     optLen := 0
76   END IpSendRequest;
77
78   TEMPLATE ArpRequest: ArpPacket =
79     (operation = 1) &
80     (senderHwAdr = (00H, 00H, 0E8H, 6BH, 0A8H, 31H)) &
81     (senderProtAdr = (146, 232, 212, 203)) &
82     (targetProtAdr = (146, 232, 212, 134))
83   END ArpRequest;
84
85   TEMPLATE ArpReply: ArpPacket :=
86     hwType := 1;
87     protType := 800H;
88     hwLen := 6;
89     protLen := 4;
90     operation := 2;
91     senderHwAdr := (00H, 10H, 4BH, 30H, 5FH, 10H);
92     senderProtAdr := (146, 232, 212, 134);
93     targetHwAdr := (00H, 00H, 0E8H, 6BH, 0A8H, 31H);
94     targetProtAdr := (146, 232, 212, 203)
95   END ArpReply;
96
97   TEMPLATE IPPacket0: IPPacket =
98     (version = 4) &
99     (IHL = 5) &
100    (Flags = 0) &
101    (FragmentOfs = 0)
102 END IPPacket0;

```

## APPENDIX C. TEST SUITE EXAMPLE

## 85

```

103
104 TEMPLATE IpSendRequest2(source, destination: IPAddress): IPSend :=
105   src := source; (* SUTIP *)
106   dst := destination; (* TSIP *)
107   prot := 0;
108   TOS := 0;
109   TTL := 32;
110   DF := FALSE;
111   Id := 1;
112   BufLen := 1500 - 20;
113   optLen := 0
114 END IpSendRequest2;
115
116 TEMPLATE ArpRequestRcv(senderEth: EthAddress; senderIP, targetIP: IPAddress): ArpPacket =
117   (operation = 1) &
118   (senderHwAddr = senderEth) &
119   (senderProtAddr = senderIP) &
120   (targetProtAddr = targetIP)
121 END ArpRequestRcv;
122
123 TEMPLATE ArpReplySnd(senderEth, targetEth: EthAddress; senderIP, targetIP: IPAddress): ArpPacket :=
124   hwType := 1;
125   protType := 800H;
126   hwLen := 6;
127   protLen := 4;
128   operation := 2;
129   senderHwAddr := senderEth;
130   senderProtAddr := senderIP;
131   targetHwAddr := targetEth;
132   targetProtAddr := targetIP
133 END ArpReplySnd;
134
135 TEMPLATE IPPacketRcv(source, destination: IPAddress): IPPacket =
136   (version = 4) &
137   (IHL = 5) &
138   (Flags = 0) &
139   (FragmentOfs = 0) &
140   (SrcAdr = source) &
141   (DstAdr = destination)
142 END IPPacketRcv;
143
144 TEMPLATE ArpRequestSnd(senderEth: EthAddress; senderIP, targetIP: IPAddress): ArpPacket :=
145   hwType := 1;
146   protType := 800H;
147   hwLen := 6;
148   protLen := 4;
149   operation := 1;
150   senderHwAddr := senderEth;
151   senderProtAddr := senderIP;
152   targetProtAddr := targetIP
153 END ArpRequestSnd;
154
155 TEMPLATE ArpReplyRcv(senderEth, targetEth: EthAddress; senderIP, targetIP: IPAddress): ArpPacket =
156   (operation = 2) &
157   (senderHwAddr = senderEth) &
158   (senderProtAddr = senderIP) &
159   (targetHwAddr = targetEth) &
160   (targetProtAddr = targetIP)
161 END ArpReplyRcv;
162
163 TEMPLATE IPPacketSnd(source, destination: IPAddress): IPPacket :=
164   version := 4;
165   IHL := 5;
166   TotalLen := 60;
167   Flags := 2;
168   TTL := 32;

```

## APPENDIX C. TEST SUITE EXAMPLE

```

169     Protocol := 0;
170     SrcAdr := source;
171     DstAdr := destination
172 END IPPacketSnd;
173
174 TEMPLATE IPDeliveryIndication(source, destination: IPAddress): IPDeliver =
175     (src = source) &
176     (dst = destination) &
177     (prot = 0) &
178     (BufLen = 40)
179 END IPDeliveryIndication;
180
181 (* ----- TEST SYSTEM INTERFACE DEFINITION ----- *)
182
183 INTERFACE
184     LTpco: [INOUT ArpPacket, IPPacket];
185     UTpco: [OUT IPSend; IN IPDeliver]
186 END;
187
188 (* ----- TEST CASES ----- *)
189
190 TESTCASE ArplpSend;
191     TIMER ReceiveTimer := 1000; (* ms *)
192     VAR ArpReceived: BOOLEAN;
193 BEGIN
194     ArpReceived := FALSE;
195     UTpco ! IpSendRequest;
196     START ReceiveTimer;
197     ALT
198     [ ~ArpReceived ] LTpco ? ArpRequest ->
199         ArpReceived := TRUE;
200         LTpco ! ArpReply;
201         REPEAT
202         [ ] LTpco ? IPPacket0 ->
203             CANCEL ReceiveTimer;
204             VERDICT PASS
205         [ ] TIMEOUT ReceiveTimer ->
206             VERDICT FAIL
207     END
208 END ArplpSend;
209
210 TESTCASE ArplpSend2;
211     TIMER ReceiveTimer := 1000; (* ms *)
212     VAR
213         sutEthAdr, testsystemEthAdr: EthAddress;
214         sutIPAdr, testsystemIPAdr: IPAddress;
215 BEGIN
216     testsystemEthAdr := (00H, 10H, 4BH, 30H, 5FH, 10H);
217     sutEthAdr := (00H, 00H, 0E8H, 6BH, 0A8H, 31H);
218     testsystemIPAdr := (146, 232, 212, 134);
219     sutIPAdr := (146, 232, 212, 203);
220
221     UTpco ! IpSendRequest2(sutIPAdr, testsystemIPAdr);
222     START ReceiveTimer;
223     ALT
224     [ ] LTpco ? ArpRequestRcv(sutEthAdr, sutIPAdr, testsystemIPAdr) ->
225         LTpco ! ArpReplySnd(testsystemEthAdr, sutEthAdr, testsystemIPAdr, sutIPAdr);
226         ALT
227         [ ] LTpco ? IPPacketRcv(sutIPAdr, testsystemIPAdr) ->
228             CANCEL ReceiveTimer;
229             VERDICT PASS
230         [ ] TIMEOUT ReceiveTimer ->
231             VERDICT FAIL
232     END
233     [ ] LTpco ? IPPacketRcv(sutIPAdr, testsystemIPAdr) ->
234         CANCEL ReceiveTimer;

```

## APPENDIX C. TEST SUITE EXAMPLE

```

235   VERDICT PASS
236   [ ] TIMEOUT ReceiveTimer ->
237     VERDICT FAIL
238   END
239 END ArplpSend2;

240
241 TESTCASE ArplpDeliver;
242   TIMER ArpReplyTimer, DeliveryTimer := 1000; (* ms *)
243   VAR
244     sutEthAddr, testsystemEthAddr: EthAddress;
245     sutIPAddr, testsystemIPAddr: IPAddress;
246 BEGIN
247   testsystemEthAddr := (00H, 10H, 4BH, 30H, 5FH, 10H);
248   sutEthAddr := (00H, 00H, 0E8H, 6BH, 0A8H, 31H);
249   testsystemIPAddr := (146, 232, 212, 134);
250   sutIPAddr := (146, 232, 212, 203);
251
252   LTpc0 ! ArpRequestSnd(testsystemEthAddr, testsystemIPAddr, sutIPAddr);
253   START ArpReplyTimer;
254   ALT
255   [ ] LTpc0 ? ArpReplyRcv(sutEthAddr, testsystemEthAddr, sutIPAddr, testsystemIPAddr) ->
256     CANCEL ArpReplyTimer;
257     LTpc0 ! IPPacketSnd(testsystemIPAddr, sutIPAddr);
258     START DeliveryTimer;
259     ALT
260     [ ] UTpc0 ? IPDeliveryIndication(testsystemIPAddr, sutIPAddr) ->
261       CANCEL DeliveryTimer;
262       VERDICT PASS
263     [ ] TIMEOUT DeliveryTimer ->
264       VERDICT FAIL
265     END
266   [ ] TIMEOUT ArpReplyTimer ->
267     VERDICT FAIL
268   END
269 END ArplpDeliver;
270
271 END ARPIPTTestSuite.

```

The ArplpSend and ArplpSend2 are two variations of the same test case. The second one makes use of test context specific variables, parameterized templates, and a nested **ALT** construct.

## C.1 Generated Module for Type Declarations

All the type declarations of the current test suite is translated to module PTSTypes:

```

1 MODULE PTSTypes; (* ARPIPTTestSuite *)
2 IMPORT U := UINTS, RT := PTSRuntime;
3
4 CONST
5   idIPSend* = 0;
6   idIPDeliver* = 1;
7   idArpPacket* = 2;
8   idIPPacket* = 3;
9
10 TYPE
11   EthAddress* = ARRAY 6 OF U.UINT8;
12   IPAddress* = ARRAY 4 OF U.UINT8;
13

```

## APPENDIX C. TEST SUITE EXAMPLE

```

14    IPSend* = POINTER TO IPSendDesc;
15    IPSendDesc* = RECORD (RT.MessageDesc)
16        src*: IPAddress;
17        dst*: IPAddress;
18        prot*: U.UINT8;
19        TOS*: U.UINT8;
20        TTL*: U.UINT8;
21        DF*: BOOLEAN;
22        Id*: U.UINT16;
23        BufLen*: U.UINT16;
24        Buffer*: ARRAY 1500 OF U.UINT8;
25        opt*: ARRAY 40 OF U.UINT8;
26        optLen*: U.UINT8
27    END;
28
29    IPDeliver* = POINTER TO IPDeliverDesc;
30    IPDeliverDesc* = RECORD (RT.MessageDesc)
31        src*: IPAddress;
32        dst*: IPAddress;
33        prot*: U.UINT8;
34        TOS*: U.UINT8;
35        BufLen*: U.UINT16;
36        Buffer*: ARRAY 1500 OF U.UINT8;
37        opt*: ARRAY 40 OF U.UINT8;
38        optLen*: U.UINT8
39    END;
40
41    ArpPacket* = POINTER TO ArpPacketDesc;
42    ArpPacketDesc* = RECORD (RT.MessageDesc)
43        hwType*: U.UINT16;
44        protType*: U.UINT16;
45        hwLen*: U.UINT8;
46        protLen*: U.UINT8;
47        operation*: U.UINT16;
48        senderHwAddr*: EthAddress;
49        senderProtAddr*: IPAddress;
50        targetHwAddr*: EthAddress;
51        targetProtAddr*: IPAddress
52    END;
53
54    IPPacket* = POINTER TO IPPacketDesc;
55    IPPacketDesc* = RECORD (RT.MessageDesc)
56        versionIHL*: U.UINT8;
57        TOS*: U.UINT8;
58        TotalLen*: U.UINT16;
59        Id*: U.UINT16;
60        FlagsFragmentOfs*: U.UINT16;
61        TTL*: U.UINT8;
62        Protocol*: U.UINT8;
63        HdrChkSum*: U.UINT16;
64        SrcAddr*: IPAddress;
65        DstAddr*: IPAddress;
66        OptionsOrData*: ARRAY 40 OF U.UINT8;
67        Data*: ARRAY 1440 OF U.UINT8
68    END;
69
70    PROCEDURE ByteOrder*(msg: RT.Message);
71    BEGIN
72        IF msg IS IPSend THEN
73            WITH msg: IPSend DO
74                RT.ByteOrder16(msg.Id);
75                RT.ByteOrder16(msg.BufLen)
76        END
77        ELSIF msg IS IPDeliver THEN
78            WITH msg: IPDeliver DO
79                RT.ByteOrder16(msg.BufLen)

```

```

80      END
81  ELSIF msg IS ArpPacket THEN
82    WITH msg: ArpPacket DO
83      RT.ByteOrder16(msg.hwType);
84      RT.ByteOrder16(msg.protType);
85      RT.ByteOrder16(msg.operation)
86    END
87  ELSIF msg IS IPPacket THEN
88    WITH msg: IPPacket DO
89      RT.ByteOrder16(msg.TotalLen);
90      RT.ByteOrder16(msg.Id);
91      RT.ByteOrder16(msg.FlagsFragmentOfs);
92      RT.ByteOrder16(msg.HdrChkSum)
93    END
94  ELSE
95    (* ***** *)
96  END
97 END ByteOrder;
98
99 END PTSTypes. (* ARPIPTestSuite *)

```

The ByteOrder procedure changes the byte order of all the 16-bit and 32-bit basic typed fields in the given message. It is accessible to a test adapter to finalize a message to be sent or a received message to be enqueued.

## C.2 Generated Module for Test Suite Behaviour

Module PTSETS contains all the procedures associated with the behaviour of the specified test suite:

```

1 MODULE PTSETS; (* ARPIPTestSuite *)
2   IMPORT S := SYSTEM, U := UINTS, Oberon, RT := PTSRuntime, T := PTSTypes;
3
4   TYPE
5     INTERFACE = RECORD
6       LTpco: RT.Port;
7       UTpco: RT.Port
8     END;
9
10    ArpRequestMS = POINTER TO ArpRequestMSDesc;
11    ArpRequestMSDesc = RECORD (RT.MatchingSpecDesc)
12    END;
13
14    IPPacket0MS = POINTER TO IPPacket0MSDesc;
15    IPPacket0MSDesc = RECORD (RT.MatchingSpecDesc)
16    END;
17
18    ArpRequestRcvMS = POINTER TO ArpRequestRcvMSDesc;
19    ArpRequestRcvMSDesc = RECORD (RT.MatchingSpecDesc)
20      senderEth: T.EthAddress;
21      senderIP: T.IPAddress;
22      targetIP: T.IPAddress
23    END;
24
25    IPPacketRcvMS = POINTER TO IPPacketRcvMSDesc;
26    IPPacketRcvMSDesc = RECORD (RT.MatchingSpecDesc)

```

## APPENDIX C. TEST SUITE EXAMPLE

90

```

27      source: T.IPAddress;
28      destination: T.IPAddress
29  END;
30
31 ArpReplyRcvMS = POINTER TO ArpReplyRcvMSDesc;
32 ArpReplyRcvMSDesc = RECORD (RT.MatchingSpecDesc)
33   senderEth: T.EthAddress;
34   targetEth: T.EthAddress;
35   senderIP: T.IPAddress;
36   targetIP: T.IPAddress
37  END;
38
39 IPDeliveryIndicationMS = POINTER TO IPDeliveryIndicationMSDesc;
40 IPDeliveryIndicationMSDesc = RECORD (RT.MatchingSpecDesc)
41   source: T.IPAddress;
42   destination: T.IPAddress
43  END;
44
45 ArplpSendTC = POINTER TO ArplpSendTCDesc;
46 ArplpSendTCDesc = RECORD (RT.TestCaseDesc)
47   i: INTERFACE;
48   state: LONGINT;
49   ReceiveTimer: RT.Timer;
50   ArpReceived: BOOLEAN
51  END;
52
53 ArplpSend2TC = POINTER TO ArplpSend2TCDesc;
54 ArplpSend2TCDesc = RECORD (RT.TestCaseDesc)
55   i: INTERFACE;
56   state: LONGINT;
57   ReceiveTimer: RT.Timer;
58   sutEthAddr: T.EthAddress;
59   testsystemEthAddr: T.EthAddress;
60   sutIPAddr: T.IPAddress;
61   testsystemIPAddr: T.IPAddress
62  END;
63
64 ArpIPDeliverTC = POINTER TO ArpIPDeliverTCDesc;
65 ArpIPDeliverTCDesc = RECORD (RT.TestCaseDesc)
66   i: INTERFACE;
67   state: LONGINT;
68   ArpReplyTimer: RT.Timer;
69   DeliveryTimer: RT.Timer;
70   sutEthAddr: T.EthAddress;
71   testsystemEthAddr: T.EthAddress;
72   sutIPAddr: T.IPAddress;
73   testsystemIPAddr: T.IPAddress
74  END;
75
76 VAR
77   dummyMsgVar: RT.Message; (* variable used to store an unused received message *)
78
79 ...
80
81
82
83
84
85 PROCEDURE ArplpSendHandler(me: Oberon.Task);
86 BEGIN
87   WITH me: ArplpSendTC DO
88     CASE me.state OF
89       0:
90         me.ArpReceived := FALSE;
91         RT.Send(me.i.UTpco, "IPSendRequest", IpSendRequest());
92         RT.StartTimer(me.ReceiveTimer);
93         me.state := 1
94       | 1:
95         IF ~me.ArpReceived & RT.Receive(me.i.UTpco, "ArpRequest", ArpRequest(), dummyMsgVar) THEN

```

## APPENDIX C. TEST SUITE EXAMPLE

```

496     me.Arpreceived := TRUE;
497     RT.Send(me.i.LTpco, "Arpreply", Arpreply());
498     me.state := 1
499   ELSIF RT.Receive(me.i.LTpco, "IPPacket0", IPPacket0(), dummyMsgVar) THEN
500     RT.CancelTimer(me.ReceiveTimer);
501     RT.SetVerdict(me, RT.PASS);
502     me.state := 2147483647
503   ELSIF RT.Timeout(me.ReceiveTimer) THEN
504     RT.SetVerdict(me, RT.FAIL);
505     me.state := 2147483647
506   END
507 ELSE
508   RT.StopTestCase(me)
509 END
510 END
511 END ArplpSendHandler;

512
513 PROCEDURE StartArplpSendTC*;
514 VAR
515   TC: ArplpSendTC;
516 BEGIN
517   NEW(TC);
518   RT.OpenPort(TC.i.LTpco, "LTpco");
519   RT.OpenPort(TC.i.UTpco, "UTpco");
520   TC.ReceiveTimer := RT.NewTimer("ReceiveTimer", 1000);
521   RT.StartTestCase(TC, "ArplpSend", ArplpSendHandler)
522 END StartArplpSendTC;
523
524 PROCEDURE ArplpSend2Handler(me: Oberon.Task);
525 BEGIN
526   WITH me: ArplpSend2TC DO
527     CASE me.state OF
528       0:
529         me.testsystemehtadr[0] := S.VAL(U.UINT8, 0);
530         me.testsystemehtadr[1] := S.VAL(U.UINT8, 16);
531         me.testsystemehtadr[2] := S.VAL(U.UINT8, 75);
532         me.testsystemehtadr[3] := S.VAL(U.UINT8, 48);
533         me.testsystemehtadr[4] := S.VAL(U.UINT8, 95);
534         me.testsystemehtadr[5] := S.VAL(U.UINT8, 16);
535         me.sutethadr[0] := S.VAL(U.UINT8, 0);
536         me.sutethadr[1] := S.VAL(U.UINT8, 0);
537         me.sutethadr[2] := S.VAL(U.UINT8, 232);
538         me.sutethadr[3] := S.VAL(U.UINT8, 107);
539         me.sutethadr[4] := S.VAL(U.UINT8, 168);
540         me.sutethadr[5] := S.VAL(U.UINT8, 49);
541         me.testsystemIPAdr[0] := S.VAL(U.UINT8, 146);
542         me.testsystemIPAdr[1] := S.VAL(U.UINT8, 232);
543         me.testsystemIPAdr[2] := S.VAL(U.UINT8, 212);
544         me.testsystemIPAdr[3] := S.VAL(U.UINT8, 134);
545         me.sutIPAdr[0] := S.VAL(U.UINT8, 146);
546         me.sutIPAdr[1] := S.VAL(U.UINT8, 232);
547         me.sutIPAdr[2] := S.VAL(U.UINT8, 212);
548         me.sutIPAdr[3] := S.VAL(U.UINT8, 203);
549         RT.Send(me.i.UTpco, "IPSendRequest2", IpSendRequest2(me.sutIPAdr, me.testsystemIPAdr));
550         RT.StartTimer(me.ReceiveTimer);
551         me.state := 1
552       | 1:
553         IF RT.Receive(me.i.LTpco, "ArprequestRcv",
554           ArprequestRcv(me.sutethadr, me.sutIPAdr, me.testsystemIPAdr), dummyMsgVar) THEN
555           RT.Send(me.i.LTpco, "ArpreplySnd",
556             ArpreplySnd(me.testsystemehtadr, me.sutethadr, me.testsystemIPAdr, me.sutIPAdr));
557           me.state := 2
558         ELSIF RT.Receive(me.i.LTpco, "IPPacketRcv",
559           IPPacketRcv(me.sutIPAdr, me.testsystemIPAdr), dummyMsgVar) THEN
560           RT.CancelTimer(me.ReceiveTimer);
561           RT.SetVerdict(me, RT.PASS);

```

## APPENDIX C. TEST SUITE EXAMPLE

```

559         me.state := 2147483647
560     ELSIF RT.Timeout(me.ReceiveTimer) THEN
561         RT.SetVerdict(me, RT.FAIL);
562         me.state := 2147483647
563     END
564 | 2:
565     IF RT.Receive(me.i.LTpco, "IPPacketRcv",
566                 IPPacketRcv(me.sutIPAdr, me.testsysteIPAdr), dummyMsgVar) THEN
567         RT.CancelTimer(me.ReceiveTimer);
568         RT.SetVerdict(me, RT.PASS);
569         me.state := 2147483647
570     ELSIF RT.Timeout(me.ReceiveTimer) THEN
571         RT.SetVerdict(me, RT.FAIL);
572         me.state := 2147483647
573     END
574     ELSE
575         RT.StopTestCase(me)
576     END
577 END ArlpSend2Handler;
578
579 PROCEDURE StartArlpSend2TC*;
580 VAR
581     TC: ArlpSend2TC;
582 BEGIN
583     NEW(TC);
584     RT.OpenPort(TC.i.LTpco, "LTpco");
585     RT.OpenPort(TC.i.UTpco, "UTpco");
586     TC.ReceiveTimer := RT.NewTimer("ReceiveTimer", 1000);
587     RT.StartTestCase(TC, "ArlpSend2", ArlpSend2Handler)
588 END StartArlpSend2TC;
589
590 PROCEDURE ArlpDeliverHandler(me: Oberon.Task);
591 BEGIN
592     WITH me: ArlpDeliverTC DO
593         CASE me.state OF
594             0:
595                 me.testsysteEthAdr[0] := S.VAL(U.UINT8, 0);
596                 me.testsysteEthAdr[1] := S.VAL(U.UINT8, 16);
597                 me.testsysteEthAdr[2] := S.VAL(U.UINT8, 75);
598                 me.testsysteEthAdr[3] := S.VAL(U.UINT8, 48);
599                 me.testsysteEthAdr[4] := S.VAL(U.UINT8, 95);
600                 me.testsysteEthAdr[5] := S.VAL(U.UINT8, 16);
601                 me.sutEthAdr[0] := S.VAL(U.UINT8, 0);
602                 me.sutEthAdr[1] := S.VAL(U.UINT8, 0);
603                 me.sutEthAdr[2] := S.VAL(U.UINT8, 232);
604                 me.sutEthAdr[3] := S.VAL(U.UINT8, 107);
605                 me.sutEthAdr[4] := S.VAL(U.UINT8, 168);
606                 me.sutEthAdr[5] := S.VAL(U.UINT8, 49);
607                 me.testsysteIPAdr[0] := S.VAL(U.UINT8, 146);
608                 me.testsysteIPAdr[1] := S.VAL(U.UINT8, 232);
609                 me.testsysteIPAdr[2] := S.VAL(U.UINT8, 212);
610                 me.testsysteIPAdr[3] := S.VAL(U.UINT8, 134);
611                 me.sutIPAdr[0] := S.VAL(U.UINT8, 146);
612                 me.sutIPAdr[1] := S.VAL(U.UINT8, 232);
613                 me.sutIPAdr[2] := S.VAL(U.UINT8, 212);
614                 me.sutIPAdr[3] := S.VAL(U.UINT8, 203);
615                 RT.Send(me.i.LTpco, "ArpRequestSnd",
616                         ArpRequestSnd(me.testsysteEthAdr, me.testsysteIPAdr, me.sutIPAdr));
617                 RT.StartTimer(me.ArpReplyTimer);
618                 me.state := 1
619 | 1:
620     IF RT.Receive(me.i.LTpco, "ArpReplyRcv",
621                 ArpReplyRcv(me.sutEthAdr, me.testsysteEthAdr, me.sutIPAdr, me.testsysteIPAdr),
622                 dummyMsgVar) THEN
623         RT.CancelTimer(me.ArpReplyTimer);

```

```

621     RT.Send(me.i.LTpco, "IPPacketSnd", IPPacketSnd(me.testsysteIPAdr, me.sutIPAdr));
622     RT.StartTimer(me.DeliveryTimer);
623     me.state := 2
624   ELSIF RT.Timeout(me.ArpReplyTimer) THEN
625     RT.SetVerdict(me, RT.FAIL);
626     me.state := 2147483647
627   END
628   | 2:
629   | IF RT.Receive(me.i.UTpco, "IPDeliveryIndication",
630     IPDeliveryIndication(me.testsysteIPAdr, me.sutIPAdr), dummyMsgVar) THEN
631     RT.CancelTimer(me.DeliveryTimer);
632     RT.SetVerdict(me, RT.PASS);
633     me.state := 2147483647
634   ELSIF RT.Timeout(me.DeliveryTimer) THEN
635     RT.SetVerdict(me, RT.FAIL);
636     me.state := 2147483647
637   END
638   ELSE
639     RT.StopTestCase(me)
640   END
641 END ArpIPDeliverHandler;
642
643 PROCEDURE StartArpIPDeliverTC*;
644   VAR
645     TC: ArpIPDeliverTC;
646   BEGIN
647     NEW(TC);
648     RT.OpenPort(TC.i.LTpco, "LTpco");
649     RT.OpenPort(TC.i.UTpco, "UTpco");
650     TC.ArpReplyTimer := RT.NewTimer("ArpReplyTimer", 1000);
651     TC.DeliveryTimer := RT.NewTimer("DeliveryTimer", 1000);
652     RT.StartTestCase(TC, "ArpIPDeliver", ArpIPDeliverHandler)
653   END StartArpIPDeliverTC;
654
655 END PTSETS. (* ARPITestSuite *)

```

Module PTSETS exports an Oberon command for every test case. The Oberon user has full control over the order of execution and the number of repetitions. The commands for this module are:

```

PTSETS.StartArpIpSendTC
PTSETS.StartArpIpSend2TC
PTSETS.StartArpIPDeliverTC

```

A test execution log is provided in the System Log of the Oberon system. Figure 32 shows a log of the last two test cases. The numbers shown in parenthesis, after the date and time, indicate the uptime of the Oberon system in milliseconds when an event was logged.

## APPENDIX C. TEST SUITE EXAMPLE

94

The screenshot shows a window titled 'System.Log' with a menu bar containing 'File', 'Edit', 'View', 'Log', 'Help', and a toolbar with buttons for 'Close', 'Copy', 'Locate', 'Search', and 'Clear'. The current time is displayed as '14:59'. The log window contains two entries of test execution logs:

```
27.10.2002 14:59:05 (124762) TESTCASE ArplpSend
27.10.2002 14:59:05 (124786) UTpco ! IpSendRequest
27.10.2002 14:59:06 (124801) START ReceiveTimer = 1000
27.10.2002 14:59:06 (124827) ArpPacket Msg Enqueued on LTpco
27.10.2002 14:59:06 (124844) LTpco ? ArpRequest
27.10.2002 14:59:06 (124858) LTpco ! ArpReply
27.10.2002 14:59:06 (124882) IPPacket Msg Enqueued on LTpco
27.10.2002 14:59:06 (124898) LTpco ? IPPacket0
27.10.2002 14:59:06 (124910) CANCEL ReceiveTimer
27.10.2002 14:59:06 (124928) VERDICT PASS

27.10.2002 14:59:13 (132153) TESTCASE ArpIPDeliver
27.10.2002 14:59:13 (132169) LTpco ! ArpRequestSnd
27.10.2002 14:59:13 (132178) START ArpReplyTimer = 1000
27.10.2002 14:59:13 (132198) ArpPacket Msg Enqueued on LTpco
27.10.2002 14:59:13 (132208) LTpco ? ArpReplyRcv
27.10.2002 14:59:13 (132215) CANCEL ArpReplyTimer
27.10.2002 14:59:13 (132222) LTpco ! IPPacketSnd
27.10.2002 14:59:13 (132229) START DeliveryTimer = 1000
27.10.2002 14:59:13 (132245) IPDeliver Msg Enqueued on UTpco
27.10.2002 14:59:13 (132252) UTpco ? IPDeliveryIndication
27.10.2002 14:59:13 (132256) CANCEL DeliveryTimer
27.10.2002 14:59:13 (132266) VERDICT PASS
```

Figure 32: A test execution log of two test cases in the Oberon System log

## Appendix D

# The PTSRuntime Module Definition

Module PTSRuntime provides the procedures needed by an executable test suite. In addition, procedures are provided to instantiate a specified test system interface and to initialize a mapping between this test system interface and the test context of an implementation under test.

### D.1 Exported Declarations

```

IMPORT Oberon, PTSQueues;

CONST
  NONE = 0; PASS = 1; INCONCLUSIVE = 2; FAIL = 3; ERROR = 4; (* Verdict Values *)
  MaxNameLen = 31;

TYPE
  Name = ARRAY MaxNameLen + 1 OF CHAR;

  TestCase = POINTER TO TestCaseDesc; (* Testcase task type *)
  TestCaseDesc = RECORD ( Oberon.TaskDesc ) END;

  Timer = POINTER TO TimerDesc; (* Timer task type *)
  TimerDesc = RECORD ( Oberon.TaskDesc ) END;

  Message = POINTER TO MessageDesc; (* Message base type *)
  MessageDesc = RECORD ( PTSQueues.ItemDesc ) END;

  Port = POINTER TO PortDesc; (* Port base type *)
  SendProcedure = PROCEDURE (port: Port; msg: Message);
  ConnectionProcedure = PROCEDURE (port: Port);
  PortDesc = RECORD END;

  MatchingSpec = POINTER TO MatchingSpecDesc; (* Matching specification base type *)
  MatchingSpecDesc = RECORD
    Match: PROCEDURE (spec: MatchingSpec; msg: Message): BOOLEAN
  END;

```

## D.2 Runtime Operations for Test Execution

```
PROCEDURE OpenPort (VAR port: Port; name: Name);  
PROCEDURE Receive (port: Port; templatename: Name; spec: MatchingSpec;  
    VAR msg: Message): BOOLEAN;  
PROCEDURE Send (port: Port; templatename: Name; msg: Message);  
PROCEDURE StartTestCase (tc: TestCase; name: Name; tchandle: Oberon.Handler);  
PROCEDURE StopTestCase (tc: TestCase);  
PROCEDURE NewTimer (name: Name; default: LONGINT): Timer;  
PROCEDURE StartTimer (timer: Timer);  
PROCEDURE CancelTimer (timer: Timer);  
PROCEDURE Timeout (timer: Timer): BOOLEAN;  
PROCEDURE SetVerdict (tc: TestCase; val: SHORTINT);
```

## D.3 Procedures used by a Test Adapter

```
PROCEDURE Reset;  
PROCEDURE AddPort (VAR port: Port; name: Name; sendProc: SendProcedure;  
    connectProc, disconnectProc: ConnectionProcedure);  
PROCEDURE EnqueueMessage (port: Port; msg: Message; typename: Name);
```

# Bibliography

- [1] Paul Baker, Ekkart Rudolph, and Ina Schieferdecker. Graphical Test Specification — The Graphical Format of TTCN-3. In *Tenth International SDL Forum*, Copenhagen, Denmark, June 2001.
- [2] Sébastien Barbin, Lénaïck Tanguy, and César Viho. Distributed Testing using a Round-robin Mechanism. In *Proceedings of the Fifth Africom International Conference on Communication Systems*, 2001.
- [3] Ana R. Cavalli, Byoung-Moon Chin, and Kilnam Chon. Testing methods for SDL systems. *Computer Networks and ISDN Systems*, 28:1669–1683, 1996.
- [4] Ana R. Cavalli, Jean Philippe Favreau, and Marc Phalippou. Standardization of formal methods in conformance testing of communication protocols. *Computer Networks and ISDN Systems*, 29:3–14, 1996.
- [5] Zhen Ru Dai, Jens Grabowski, and Helmut Neukirchen. Real-Time Test Specification with TTCN-3 (extended abstract). In *Proceedings of the Eleventh GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems*, Bruchsal, Germany, June 2001.
- [6] René de Vries and Jan Tretmans. On-the-fly Conformance Testing Using SPIN. In *Fourth Workshop on Automata Theoretic Verification with the SPIN Model Checker*, pages 115–128, 1998.
- [7] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir. Test development for communication protocols: towards automation. *Computer Networks*, 31:1835–1872, 1999.
- [8] ETH Zürich. ETH Oberon. <http://www.oberon.ethz.ch/>.

- [9] ETSI. Testing and Test Control Notation (TTCN).  
<http://www.etsi.org/ptcc/ptccttcn3.htm>.
- [10] ETSI ES 201 873-1. Methods for testing and specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 core language. V2.2.0. ETSI, May 2002.
- [11] ETSI ES 201 873-3. Methods for testing and specification (MTS); The Testing and Test Control Notation version 3; Part 3: Graphical Presentation Format for TTCN-3 (GFT). V2.2.0. ETSI, March 2002.
- [12] ETSI TR 101 666. Information technology — Open Systems Interconnection Conformance testing methodology and framework; The tree and tabular combined notation (TTCN) (Ed. 2++) V1.0.0. ETSI, May 1999.
- [13] ETSI TR 102 043. Methods for testing and specification (MTS); The TTCN-3 runtime interface (TRI); Concepts and definition of the TRI. V1.1.1. ETSI, April 2002.
- [14] Susumu Fujiwara, Gregor v Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderazak Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on software engineering*, 17(6):591–603, June 1991.
- [15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 26–28, 139–150. Addison-Wesley, 1995.
- [16] Jens Grabowski, Dieter Hogrefe, Robert Nahm, and Andreas Spichiger. Relating Test Purposes to Formal Specifications: Towards a Theoretical Foundation of Practical Testing. Technical Report IAM-93-014, University of Berne, Institute for Informatics, Berne, Switzerland, June 1993.
- [17] Fred Halsall. *Data Communications, Computer Networks and OSI*, chapter 6. Addison-Wesley, 1988.
- [18] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [19] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [20] ITU-T Recommendation X.290. OSI Conformance Testing and Framework for Protocol Recommendations for ITU-T Applications — General Concepts. ITU-T, April 1995.
- [21] ITU-T Recommendation X.291. OSI Conformance Testing and Framework for Protocol Recommendations for ITU-T Applications — Abstract Test Suite Specification. ITU-T, April 1995.

- [22] ITU-T Recommendation Z.500. Methods for validation and testing; Framework on formal methods in conformance testing. ITU-T, May 1997.
- [23] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, 1995.
- [24] Finn Kristoffersen and Thomas Walter. TTCN: Towards a formal semantics and validation of test suites. *Computer Networks and ISDN Systems*, 29:15–47, 1996.
- [25] Sonja Mayer. The Interface Concept within Ericsson's SCS Test System in Comparison to GCI/TRI and Prototypical Implementation of an Adapter. Diplomarbeit, Institut für Telematik, Medizinische Universität zu Lübeck, 2001.
- [26] Susan C. Murphy, Per Gunningberg, and John P.J. Kelly. Experiences with Estelle, LOTOS and SDL: a protocol implementation experiment. *Computer Networks and ISDN Systems*, 22:51–59, 1991.
- [27] W.D. Myburgh. A Test Coverage Analyser for Software Testing. Honours Project Report, University of Stellenbosch, 2000.
- [28] Gerald Neufeld and Son Vuong. An Overview of ASN.1. *Computer Networks and ISDN Systems*, 23:393–415, 1992.
- [29] OpenTTCN. OpenTTCN Tester.  
<http://www.openttcn.com/Sections/Products/ProductTester/>.
- [30] J. Peleska and M. Siegel. Test Automation of Safety-Critical Reactive Systems. *South African Computer Journal*, 19:53–77, 1997.
- [31] R.L. Probert and O. Monkewich. TTCN: the International Notation for Specifying tests of Communications Systems. *Computer Networks and ISDN Systems*, 23:417–438, 1992.
- [32] D. Rayner. A System for testing protocol implementations. In *Proceedings of the IFIP WG6.1 Second International Workshop on Protocol Specification, Testing and Verification*, pages 539–554. North-Holland Publishing Company, 1982.
- [33] D. Rayner. OSI Conformance Testing. *Computer Networks and ISDN Systems*, 14:79–98, 1987.
- [34] Martin Reiser. *The Oberon System: User Guide and Programmer's Manual*. Addison-Wesley, 1991.
- [35] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison-Wesley, 1992.

- [36] Behçet Sarikaya. Conformance Testing: Architectures and Test Sequences. *Computer Networks and ISDN Systems*, 17:111–126, 1989.
- [37] Ina Schieferdecker and Jens Grabowski. Conformance testing with TTCN. *Telektronikk*, 96(4):85–95, 2000.
- [38] Ina Schieferdecker, Stephan Pietsch, and Theofanis Vassiliou-Goiles. Systematic testing of internet protocols: First experiences in using TTCN-3 for SIP. In *Proceedings of the Fifth Africom International Conference on Communication Systems*, 2001.
- [39] Robin Sharp. *Principles of Protocol Design*. Prentice Hall, 1994.
- [40] Deepinder P. Sidhu and Ting-Kau Leung. Formal Methods for Protocol Testing: A Detailed Study. *IEEE Transactions on software engineering*, 15(4):413–426, April 1989.
- [41] William Stallings. *Data and Computer Communications*. Prentice-Hall International, fifth edition, 1997.
- [42] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*. Addison-Wesley, 1994.
- [43] Marine Tabourier, Ana Cavalli, and Melania Ionescu. A GSM-MAP protocol experiment using passive testing. In J. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods 1999, Vol. I, LNCS 1708*, pages 915–934, 1999.
- [44] Telelogic AB. Telelogic Tau TTCN Suite.  
<http://www.telelogic.com/products/tau/>.
- [45] R. Terpstra, L. Ferreira Pires, L. Heerink, and J. Tretmans. Testing Theory in Practice: A Simple Experiment. In T. Kapus and Z. Brezočnik, editors, *COST 247 Int. Workshop on Applied Formal Methods in System Design*, pages 168–183, Maribor, Slovenia, 1996. University of Maribor. Also: Technical Report No. 96-21, Centre for Telematics and Information Technology, University of Twente, The Netherlands.
- [46] Testing Technologies. TT Tool Series.  
<http://www.testingtech.de/products/TTToolSeries.html>.
- [47] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7<sup>th</sup> European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.
- [48] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

- [49] Hasan Ural. Test sequence selection based on static data flow analysis. *Computer Communications*, 10(5):234–242, October 1987.
- [50] Gregor v Bochmann, Rachida Dssouli, and J.R. Zhao. Trace analysis for conformance and arbitrating testing. *IEEE Transactions on software engineering*, 15(11):1347–1356, November 1989.
- [51] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.