Using FDT LOTOS to derive tests

by

Martijn A.M. ten Hacken


in charge of C.J. Koomen
Related to

Eindhoven University of Technology
Faculty of Electrical Engineering
Department Digital Systems
Den Dolech 2
5600 BM Eindhoven
The Netherlands
+31.40.479111

under daily supervision of Ph. Guillet
Related to

TRT
(Télécommunications Radioélectriques et Téléphoniques)
5, Avenue Réaumur - B.P. 21
92352 Le Plessis Robinson Cedex
France
+33.1.40.94.70.00

The department of electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student project and graduation reports.
Summary

From the first of December until the 14th of July, my work toward graduating at the Eindhoven University of Technology, was fulfilled within the company TRT, Paris. TRT is a business unit of Philips' division Communication Systems.

The purpose of this project was to improve the test environment of TRT, by using formal methods. To achieve this goal the Formal Description Technique (FDT) LOTOS was used to specify the behaviour of protocols. These protocols are tested after they have been implemented. In fact the specification drawn up is a specification of the implementation which will be tested. This specification will be used for automatic generation of these tests. As an example a part of the X25 LAPB protocol is specified to derive these tests.

From the specification in LOTOS an Extended Finite State Machine (EFSM) is derived, by using the calculation facilities that exist in the Language LOTOS. Also a tool called SMILE is used to generate this EFSM. This EFSM could also be derived from a specification in another language.

On the base of this EFSM a method, which derives one test suite consisting of one test trace for every input action of the Implementation Under Test (IUT), is found. Such a test trace is divided in four parts:
- initialization, to arrive in the state before the action can be transmitted.
- evaluation, to send the action tested and to receive the reactions
- verification, to verify if the state reached is correct.
- termination, to arrive at the initial state, so the test cases can be concatenated to one test suite.

A derived test case will be written in the language TTCN, which is common used for test descriptions. This language corresponds to the language used by TRT's testing machine CHAMELEON 32, but can also be used for other machines.

The verification is based on finding a unique path for the state verified, so it can be shown that the IUT has arrived in the correct state. This method is semi-formal, because you can't prove the existence of this unique path.

Finally the algorithm explained is implemented in a tool called the 'Test Suite Generator'. This tool can generate a test suite containing all the test cases with the name and states of the transaction which is tested. The tool shows that it is possible to implement the derivation method, but it has to be improved to be used in practice.
# Contents

Summary ........................................................................... i

Contents ......................................................................... ii

List of Figures and Tables ................................................ v

1 Introduction ................................................................. 1

2 TRT and the project specification ................................... 3
   2.1 An introduction to TRT ........................................... 3
      2.1.1 TRT Software Engineering Department ............... 3
   2.2 My task within TRT .............................................. 4
   2.3 Development procedure used within TRT ................. 4
   2.4 The main problem ............................................... 5
   2.5 The build up of my graduation project ...................... 5

3 Specification language LOTOS ...................................... 9
   3.1 Introduction ....................................................... 9
   3.2 CCS ................................................................ 9
   3.3 Basic LOTOS ................................................... 12
   3.4 Value Passing .................................................... 13
   3.5 LOTOS Specification Styles ................................. 14
   3.6 LOTOS tools .................................................... 16
   3.7 Testing ............................................................. 16
   3.8 Conclusions ..................................................... 16

4 Protocol specification examples ..................................... 19
   4.1 Introduction ....................................................... 19
   4.2 Protocol 4 ......................................................... 19
      4.2.1 Specification of the DCE ............................... 20
      4.2.2 The specification of the Channel .................... 23
      4.2.3 The specification of the DTE ......................... 23
Contents

4.2.4 Calculation of the parallel behaviour ........................................ 23
4.3 Translation into LOTOS ................................................................. 28
4.4 The LOTOS specification of the LAPB protocol .............................. 30
  4.4.1 LAPB Commands ............................................................... 31
  4.4.2 Information transfer phase .................................................. 32
  4.4.3 Type and Variable definition part ....................................... 34
  4.4.4 The LAPB information transfer behaviour part ....................... 35
4.5 Specification of Connection Disconnection Processes ..................... 36

5 Test suite derivation ........................................................................ 41
  5.1 Introduction ................................................................................. 41
  5.2 The Testing Block ....................................................................... 41
  5.3 The method used ........................................................................ 43
    5.3.1 Translation into an EFSM. .................................................. 44
    5.3.2 Derivation of the test suite ............................................... 46
  5.4 The pros and cons of this method .............................................. 51

6 Tools used and developed ................................................................ 53
  6.1 Introduction ................................................................................. 53
  6.2 SMILE transformation ................................................................. 53
  6.3 Test suite generator .................................................................... 54
  6.4 Used type definitions .................................................................. 55
  6.5 Module: Read EFSM .................................................................. 59
  6.6 Module: Generate test suite ....................................................... 60
  6.7 Module: Output ........................................................................... 62
  6.8 Results ....................................................................................... 63

7 Conclusions ..................................................................................... 65

Reference List ..................................................................................... 67
Acknowledgement

My intention to go abroad for a period of time became true in december 1991. To get some working experience in a foreign country was important to me. Especially because the graduation subject which is described in this report interested me extraordinarily. Also, some side aspects like learning French, meeting some foreigners and living in the city of Paris was very enjoyable. Without the help of prof. C.J. Koomen this had never been possible. I want to thank him for creating this opportunity.

Furthermore my daily supervisor Philippe Guillot helped me to find my way in the company. Unfortunately he was not there, when I finished my work within TRT. His deputy Benoit Pinta supported me during my final presentations. With their help I could create some enthusiasm in the company, so resulting in the fact that they will try to bring my work in operation.

During my work in France, Ron Koymans who is working for the Philips Research Laboratories Eindhoven followed my project. In this period we met each other a few times to have some useful discussions and to correct my report. I Would like to thank him for this. Finally I want to thank the other trainees at TRT for their help, especially the translation of some transparencies into French and the preparation of this presentation.
List of Figures and Tables

Figures:

Figure 1. The software design life cycle within TRT. ........................................ 7
Figure 2. The handshake system. ........................................................................ 11
Figure 3. Specification of a transparent communication channel ..................... 20
Figure 4. Decomposition of the transparent channel ......................................... 21
Figure 5. Scheme of the protocol 4 processes ................................................. 22
Figure 6. The structure of the LAPB protocol .................................................. 31
Figure 7. The construction of the DTE of the LAPB protocol ......................... 33
Figure 8. The structure of the Information Transfer Phase processes. .............. 34
Figure 9. The state machine of the Connection Disconnection LAPB protocol .... 39
Figure 10. A Model of the tester used within T.R.T. ........................................ 42
Figure 11. Life Cycle for testing purpose. ....................................................... 43
Figure 12. C Pointer Structure used to contain an EFSM .............................. 56

Tables:

Table 1. LAPB commands and responses ......................................................... 32
1 Introduction

The telecommunication industry has been going through a huge development in the last few decennia. Communication by computer, fax, phone, radio and television has been improved enormously. Especially when the new ISDN systems are integrated within the present networks. The future for these complex systems looks very promising.

These systems that communicate using very complex protocols call for better methods to specify the recommendations. For this matter some formal description techniques (FDTs) like LOTOS and Estelle have been developed. These new languages provide an unambiguous specification method of the protocol requirements.

Using the created calculation aspects of these languages, it is possible to derive test scenarios directly from the formal specification. These scenarios can be used to verify an implementation of a protocol. Unfortunately these methods are too modern to purchase tools that are able to derive such tests. The specification languages LOTOS and Estelle were developed one decennium ago and are ISO standards since 1989.

The purpose of this project was to propose a method for the derivation of these test scenarios. This work was done at TRT in Paris with the ambition to graduate from the Eindhoven University of Technology within the faculty Electrical Engineering and the department of digital systems.

This report is divided in two parts. This part contains the actual report and the other part consists of the appendices with the specifications written and derived, the tool
2 TRT and the project specification

2.1 An introduction to TRT

TRT is a subsidiary of Philips' Product Division Communication Systems. This division has a sales volume of 10.2 billion francs and it employs about 15,000 people.

TRT combines a wide range of technologies, systems and services to meet the communication requirements of the public and private sector. The Product division Communication systems is composed of six "Business Units" (BU) which is fully assumed by the following main subsidiaries:

- TRT - Télécommunications Radioélectriques et Téléphoniques in France
- Philips Radio Communication Systems Ltd (PRCS) in Great-Britain
- Philips Kommunikations Industrie AG (PKI) in Germany
- Philips Telecommunicatie en Data Systemen Nederland BV (PTDSN) in the Netherlands

Each "Business Unit" has a worldwide responsibility, within its allocated activity scope, for: strategy, marketing, development, manufacturing, sales, maintenance and training. One BU can be subdivided into a number of International Product Centres (IPC).

TRT sells its own items of equipment in France as well as the products from other companies of the division. On an international level, TRT sells its products either directly or through the Philips National Sales Organisations (NSO) (about forty worldwide).

2.1.1 TRT Software Engineering Department

As of 1980, TRT started a software engineering group, with the purpose to improve software design methods. To support software development they used some tools like GEODE (a development tool based on the specification language SDL). Other tools, like Platine 2, were developed for this goal. Platine 2 provides a structured software development method, which follows a predefined life cycle.

Nowadays, the software engineering department exists of about fifteen engineers and
trainees. The main project of this group is to develop a new tool platine 3, which defines an environment for software development. Platine 3 will help the developer to follow a structured path of developing given by platine's design life cycle. Platine 3 makes it possible to integrate in principle every tool in this platform. It is based on EAST framework which use the PCTE (Portable Common Tool Environment) standard.

For keeping an advantage in software engineering, Some preliminary studies are done in the CASE (Computer Aided Software Engineering) environment.


2.2 My task within TRT

In brief the description of my graduation project is: "How to apply techniques as described in "The design of communicating systems" + related techniques (e.g. LOTOS) in supporting or improving the test environment for DAC at TRT aimed at X25 or ATM protocols?". To answer this question, the development procedure which is used by DAC is explained. Further on, the main testing problem which has to be improved within DAC will be pointed out. This will finally lead to the time schedule of my graduation project.

2.3 Development procedure used within TRT.

To explain the development procedure [GUI], the description of the stages used are shown in figure 1 on page 7. 'Requirements' is a description of the product, which the client wants to purchase. 'System Specification' is the description of the product that is agreed upon to produce. These stages together are called Pre-design. The next stage is the 'Software Specification'. In this stage the specifications of the software (DOC190) and the way of how this system can be verified (DOC170) will be written. This is a specification of the software of the product which will be developed. Normally this is written in french or english.

After this phase, the specification made in document 190 will be implemented. First of all a 'Preliminary design' will be made to define the architecture of a system. This architecture is usually written in SDL and/or in an executable real time system. In block 'Detailed
Design' the structure made will be refined and the algorithms used will be selected. In the block 'Coding' the program in for example C or Pascal will be written. The software is ready for testing and integration in the blocks 'Unit Testing' and 'Integration'. In the stages 'Preliminary Design' through to 'Integration', it is possible to use development programs like GEODE, which can produce documents 161 and 160 automatically, so the software can be tested by TESTEUR, LOGISCOPE etc.. GEODE is based on the language SDL.

The verification of the whole system will be done in the block 'Verification'. The alpha tests, which are written in document 170 will be executed to test the system. For this purpose, within TRT, they use a machine called 'CHAMELEON'. Afterwards there will follow a 'Validation' block, wherein the system will be launched on the market as a beta-test version to let the client test the system.

2.4 The main problem.

Because the specifications in DOC 170 and DOC 190 are not written in a formal language, it is difficult to test the software without being incomplete and insufficient. There is a testing machine 'CHAMELEON' which can test the software protocols. This testing system uses its own programming language. Normally it takes a long time to let a product pass this phase (approximately 6 month), because writing the input file for the Chameleon machine involves a lot of work. An option to reduce this time is writing DOC 190 in a formal language and after this formalization, to create a method to automatically derive DOC 170. This document has to be written in a language which is familiar with the chameleon language, so it can be used as input file for the Chameleon tester. In this manner the passage time will be significantly reduced. The testing language which is used has to be independent of the Chameleon language, because there are other testing systems used within the PHILIPS company.

2.5 The build up of my graduation project

To reach the goal of improving the test quality or reducing testing time, my industrial training time is divided in 3 parts:

a) As an example, a part of the protocol named X25 LAPB, will be specified in a formal language. In principle this can be any formal language like SDL, Estelle or
LOTOS.

b) After the description of the protocol, a method has to be found to automatically derive the test traces from the specification. Tools which could be helpful in this stage have to be found.

c) Finally out of the formalization which is made for a part of the X25, LAPB protocol, the test traces will be derived. This becomes an example of the procedure which can be followed to specify the software and how to derive tests from this specification, with the main goal: to reduce the testing time, and thus to reduce the development life cycle time.

Two months were planned for each stage. Afterwards one and a half month were left for finishing up the work and writing a report. In this report no distinction is made between this three parts, but respectively an introduction to LOTOS, some LOTOS examples, a derivation method and a tool developed is described. The report is terminated with conclusions and recommendations.
3 Specification language LOTOS

3.1 Introduction

LOTOS (Language of Temporal Ordering Specification) [ISO3, BOL, CLA, DIA, DRA] is one of the two Formal Description Techniques (FDT) developed within ISO (International Organization for Standardization) for the formal specification of open distributed systems, and in particular for those related to the Open System Interconnection (OSI) computer network architecture. The language was developed in the 1980s and became an ISO standard in 1989. The basic idea behind LOTOS is that systems can be specified by defining the temporal relation among the interactions that constitute the externally observable behaviour of a system. In LOTOS a system is seen as a set of processes which interact and exchange data with each other and with their environment. The language is split in two parts:

1. A part for the description of data structures and value expressions. This part is based on the formal theory of abstract data types. It is inspired by the abstract data type formalism "ACT ONE".

2. A part for the description of process algebra which is based on CCS (Calculus of Communicating Systems) and CSP (Language of Communicating Sequential Processes).

This chapter contains the following sections. Section 2 introduces the basic elements of the underlying language CCS. In section 3 an introduction to basic LOTOS is made. This is the subset of LOTOS where processes interact with each other by pure synchronization, without exchanging values. In section 4 value passing and conditions are added. Section 5 gives a summation of specification styles. Section 6 gives some information about tool support. The chapter will be ended with an introduction to testing in section 7 and afterwards the conclusions will be drawn in section 8.

3.2 CCS
The Calculus of Communicating Systems (CCS) [MIL, KOO] is a formal mathematical framework which can be used to specify communication systems and to verify their properties. In CCS we can define a communication system as a finite state machine containing objects of behaviour, called agents and states. Proofs basically consist of the application of laws which are used as rewrite rules, what means rewriting the specification in an other equivalent form. CCS is founded on the two central key words: "observation" and "communication". In this language rules are defined to prove the following theorems:

1. Two systems are observation equivalent if we can’t tell them apart without pulling them apart.

2. A corresponding non parallel system can be calculated from a system with parallel processing and internal communication.

With this language it is possible to prove with mathematical rules the equivalence between the specification made and the system implemented. This language forms the Boolean algebra of communication systems.

In CCS we introduce the following set of actions:

names \((a, b, c, \ldots)\)

the complementary co-names \((\overline{a}, \overline{b}, \overline{c} \ldots)\)

silent action \(\tau\)

Suppose we have two agents A and B with A a port \(a\) and B a port \(\overline{a}\). These ports can communicate with each other because they are complementary. The resulting combination of A and B performs a silent or internal action \(\tau\). Later on we will discuss an example.

In CCS we introduce the following set of agent expressions

\[
\begin{align*}
    a &: E & \text{Action Prefix} \\
    E_1 + E_2 & & \text{Summation} \\
    E_1 | E_2 & & \text{Composition} \\
    E \mid L & & \text{Restriction} \\
    E[f] & & \text{Relabelling (f is a relabelling function)}
\end{align*}
\]

To explain this expressions and the calculation possibilities we take a handshake example. This example is shown in figure 1.
The specification of the agents A and B can be written as:

\[
\begin{align*}
A &= a.\text{in1}?: A_1 \\
A_1 &= a.\text{out}!: A_2 \\
A_2 &= a.\text{in2}?: A \\
B &= b.\text{in}?: B_1 \\
B_1 &= b.\text{out1}!: B_2 \\
B_2 &= b.\text{out2}!: B
\end{align*}
\]

To let the agents communicate we will connect A and B as:

\[
\begin{align*}
a.\text{out} &= b.\text{in} \\
a.\text{in2} &= b.\text{out2}
\end{align*}
\]

In order to calculate the behaviour of the total system C, we have to evaluate the following expression:

\[
C = (A[a/\bar{a}.\text{out}!, \bar{\alpha}/a.\text{in2}?
] \mid B[\alpha/b.\text{in}?, \bar{\beta}/\bar{b}.\text{out2}!]) \setminus \alpha, \beta
\]

Which means: C is the parallel composition of A and B, with a relabelling of a.out, b.in2, b.in and b.out so that they will communicate as given in figure 1. At last they are restricted from the environment so we can’t observe them.

This expression will allow only the set of actions \{a.\text{in1}?, b.\text{out1}!, τ\}, because the rest of the actions are restricted from the environment. The initial state of C will be AB (the paired states of A and B). The expansion algorithm defines if there is a possible action in A or in B which is not restricted then this will also be possible in C. If there is an internal communication possible between complementary names of A and B then there will occur an internal action τ in C. The expansion will give the following result:
AB = a.in1?: A1B (action b.in?: B1 is not possible because it is restricted)
A1B = τ: A2B1 (internal action between a.out! and b.in?)
A2B1 = b.out1!: A2B2
A2B2 = τ: AB (internal action between a.in2? and b.out2!)

This result is exactly what we expected, because this is the purpose of a handshake system.

We didn't see the summation operator. As an example we have the following expression:

A = a.in?: A1 + a.in3?: A3

This means that apart from executing the action a.in?, we can also execute a.in3? and continue in state A3. The next expression:

A = τ:A1 + τ:A3

could also be a CCS expression. A specification which contains an expression like this will internally decide in which state it will be set. We call this non-determinism. Normally there are more possibilities to implement a system and we don't want to choose one in the specification phase. In this case we get more possible next states and therefore non-determinism.

3.3 Basic LOTOS

The main difference between CCS and LOTOS is that LOTOS is Process oriented. A process is comparable with an agent in CCS, but there are more possibilities. A system can be viewed as one process which, will be refined into subprocesses. The system can thus be viewed as a hierarchy of interacting processes.

Basic LOTOS has almost the same semantics as CCS, only the syntax is changed. We can make the following operator transformations:

<table>
<thead>
<tr>
<th>CCS</th>
<th>LOTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: (action)</td>
<td>a; (port)</td>
</tr>
<tr>
<td>E1 \ E2</td>
<td>E1</td>
</tr>
<tr>
<td>E1 | E2</td>
<td>E1</td>
</tr>
<tr>
<td>E\L</td>
<td>hide L in E</td>
</tr>
<tr>
<td>E[f]</td>
<td>within process call</td>
</tr>
</tbody>
</table>
In LOTOS we don’t use relabelling, but we give the port names in the process calls. For example we can specify the handshake system in LOTOS in the following way:

```
process C[a_in1, b_out1]: noexit:=
  hide alpha, beta in
  A[a_in1, alpha, beta] || B[alpha, b_out1, beta]
  with
  process A[a_in1, a_out, a_in2]: noexit:=
    a_in1; a_out; a_in2; A[a_in1, a_out, a_in2]
  endproc
  process B[b_in, b_out1, b_out2]: noexit:=
    b_in; b_out1; b_out2; B[b_in, b_out1, b_out2]
endproc
endproc
```

Or shorter with extensive use of relabelling:

```
process C[a_in1, b_out1]: noexit:=
  hide alpha, beta in
  A[a_in1, alpha, beta] || A[b_out1, beta, alpha]
  with
  process A[a, b, c]: noexit:=
    a; A[b, c, a]
endproc
endproc
```

Furthermore there are some extra operators, for example the \& pure interleaving operator, which means the processes with corresponding ports will not communicate with each other but only with the environment.

### 3.4 Value Passing

In full LOTOS we can define types which we can use in our behaviour part. The definition part is not handled here, it is written in [BOO, DRA]. Here we will only discuss some principles of value passing in LOTOS’ behaviour part.

In LOTOS, you can’t speak of input and output, but only of synchronization. To pass a value to an other process, an exclamation mark is added and the value to the port name. If you want to receive a value from a port a question mark is added and the variable and its type. For example:
Pass a variable x via port a:
Pass a constant 3 via port a:
Reception of a variable of type int via port a:

\[ a!x; \]
\[ a!3; \]
\[ a?y: \text{int}; \]

The actions will synchronize and effectuate the value pass if the output variable is of the type int. In LOTOS it is also possible to synchronize the two first actions if \( x = 3 \). It is also possible to synchronize between two question marks, whereafter their values will be equalized.

If variables are defined, they can be used as parameters for a process and as predicate for an action. The following example shows these possibilities:

```plaintext
process timer[start,stp,ring,error] (tries:nat):noexit:=
  \[ \text{[tries ne N2]} \rightarrow \text{start;} \]
  \[ (\text{stp}; \text{timer[start, stp, ring, error]}(0)) \]
  \[ i;\text{ring; timer[start, stp, ring, error]}(\text{tries} +1) \]
  \[ \text{timer[start, stop, ring, error]}(\text{tries}) \]
  \[ \text{[tries eq N2]} \rightarrow \text{error!timeout; stop} \]
endproc timer
```

This process will give an error message and stop if N2 times a time out is arrived. The condition \[ \text{[tries eq N2]} \] will take care of this.

### 3.5 LOTOS Specification Styles

In LOTOS 4 specification styles are defined: constraint oriented, resource oriented, monolithic oriented and state oriented [EIJ]. We will see that the style of specifying is dependent on where you are in the design cycle. We will first introduce them with an example and afterwards we will show the differences.

**Constraint oriented**

In a constraint oriented specification, every design constraint is expressed as a separate process. So if you have to execute different functions in your protocol, for every function we use an other process and we chain them in parallel. In this sort of specification we make extensive use of parallelism and we don't use hiding. An example of this specification style is the following:
process QA_service(Q,A): noexit=
   (QA_local[Q] || QA_local[A] || QA_remote[Q,A])
where
process QA_local[X]: noexit=
   X?x:question; X?y:answer; stop
endproc
process QA_remote[X,Y]: noexit=
   X?x:question; Y!x; stop || Y?y:answer; X!y; stop
endproc
endproc

In this system the local constraints are clearly separated from the remote constraints.

Resource oriented.
Resources are parts from which a system is constructed. A resource oriented description therefore shows internal structure. The resources can for example correspond to protocol entities or parts thereof. In this style we describe a system as a composition of some resources with their communication hidden from external observation. In the example we see 4 processes receive_data, send_buffer, send_data and timer as resources and their internal communication over 8 channels. This process is a part of a X25 LAPB specification. Every resource itself can be described in any specification style (Not displayed here).

process information_transfer[prt_in, prt_out, info_in, info_out,error]:noexit:=
   hide poll,reject,confirm,resend,ack,start,stop,ring in
   receive_data[prt_in, info_out,poll,reject,confirm,resend,ack]
      |[poll,reject,confirm,resend,ack] |
      ((send_buffer[info_in,ack,resend,frame,start,stop,error] 
      |[frame] |
      send_data[prt_out, frame,poll,reject,confirm,ring,start,error])
         |[start,stop,ring] |
      timer[start,stop,ring,error](0))
   etc.

The ports mentioned in the square brackets are the ports on which the processes synchronize.

Monolithic and state oriented.
These specification styles do not show a structuring based on parallelism, but it is respectively based on event and state sequencing. Specifications in these styles are close to the structure of protocol entity implementations and also protocol standards. No parallelism and thus no hiding is used.
LOTOS Design Process
The following thesis will point out how LOTOS can be used within the design process: "An idealised design process consists of transforming a constraint oriented service specification into a resource oriented protocol specification, of which the appropriate parts are transformed into a monolithic or state oriented implementation (specification)."

Because LOTOS gives the possibility to prove observation equivalence between the different specification stages, you will make less errors and in case of a mistake it is easier found.

3.6 LOTOS tools

At this moment several toolsets are developed [WIN, LOG]. One of them is LITE [BOO], which is developed in the ESPRIT project LOTOSPHERE. At this moment it contains a syntax checker CRIE and a simulator SMILE. These tools are evaluated within the Philips Research Laboratory Eindhoven. Results of this evaluation are to be found in [LAO].

SMILE and CRIE still contain several bugs, but the main results are satisfying. The tools can handle full LOTOS with some small restrictions or sometimes with addition of some extra information, for defining variables or operators. In the future the toolset LITE [BOO] will be expanded with a graphical editor, a test generator [ALD, PAV] a compiler and an equivalence verifier.

3.7 Testing

The main purpose for my work was using formal methods for testing. For this purpose the language LOTOS is used. In brief, the method used translates a LOTOS specification in an extended finite state machine (EFSM) and from this EFSM the test suite will be derived [GUE]. The explanation and results are published in the next chapters. There are also other possibilities to derive test suites as you can read in [NAI, BRI]. They will probably be supported by tools.

3.8 Conclusions

LOTOS can be helpful to specify a system and to check every design step which is made in the life cycle. Because of the calculation aspect of LOTOS it will also be possible to
derive test traces automatically. At this moment the tool support is not fully developed, but in the near future there will be useful tools on the market. With this support the life cycle time can be significantly reduced.
4 Protocol specification examples

4.1 Introduction

In order to become familiar with formal specifications, we are going to look at another more complicated example to specify. The chosen example is described in [TAN] protocol 4 on page 269. It is the starting point, on which the LAPB protocol is based. It is a protocol which provides a full duplex communication between two points. The system contains a one place buffer, which has to be acknowledged, before a new frame can be sent. Protocol 4 ensures one of a reliable communication over an unreliable channel without losing or doubling frames. This protocol has the disadvantage of low efficiency, especially in case of long transmission times. This disadvantage is solved in the more complicated LAPB protocol, which will be explained later on in this chapter. We will first specify protocol 4 in CCS and later on translate it into LOTOS.

4.2 Protocol 4

A communication protocol provides a transparent channel. The data that is transported has to be equal to the data that is received on the other side of the channel, as shown in figure 3. The specification in CCS would be:

\[
\begin{align*}
R0 & = \text{DCE}_\text{info}_\text{in}??\text{data} : R1 + \text{DTE}_\text{info}_\text{in}??\text{data} : R2 \\
R1 & = \text{DTE}_\text{info}_\text{out}!\text{data} : R0 + \text{DTE}_\text{info}_\text{in}??\text{data} : R3 \\
R2 & = \text{DCE}_\text{info}_\text{in}??\text{data} : R3 + \text{DCE}_\text{info}_\text{out}!\text{data} : R0 \\
R3 & = \text{DTE}_\text{info}_\text{out}!\text{data} : R2 + \text{DCE}_\text{info}_\text{out}!\text{data} : R1 \\
\end{align*}
\]

which is the parallel composition \((S \parallel T)\) of the two independent processes \(S\) and \(T\)

\[
\begin{align*}
S0 & = \text{DCE}_\text{info}_\text{in}??\text{data} : S1 \\
S1 & = \text{DTE}_\text{info}_\text{out}!\text{data} : S0 \\
\end{align*}
\]

and

\[
\begin{align*}
T0 & = \text{DTE}_\text{info}_\text{in}??\text{data} : T1 \\
T1 & = \text{DCE}_\text{info}_\text{out}!\text{data} : T0 \\
\end{align*}
\]
Figure 3. Specification of a transparent communication channel

To specify protocol 4, we have to refine and decompose this specification. The first step made was dividing the system into three parts, as shown in figure 4. In principle some knowledge is added to the system, which is:

"The system is built upon a bidirectional channel and two sender/receivers called a DCE and a DTE"

It is possible to write the specification at this level, with this amount of processes, but because later on we want to specify a much more difficult protocol, we are making an extra decomposition step, as shown in figure 5. The added knowledge is:

"The DTE is divided in a sending and reception part and the DCE also contains a timer, besides these two parts"

As explained in [TAN], only the DCE has a timer, because otherwise double sent frames can appear.

4.2.1 Specification of the DCE
The processes of the DCE can be specified as follows:

receive data DCE:
R0(exp) = DCE_prt_in?data?nr?piggyback
          : R1(exp,data,nr,piggyback)
R1(exp,data,nr,piggyback) = DCE_send!piggyback: R2(exp,data,nr)
R2(exp=nr,data) = DCE_info_out!data: R3(succ(exp))
R2(exp<>nr,data) = R3(exp)
Figure 4. Decomposition of the transparent channel

R3(exp) = DCE_ack!nr: R0(exp)

send data DCE:
S0(send,data,ack) = DCE_send?nr: S1(send,data,ack,nr)
+ timeout: S3(send,data,ack)
S1(send=nr,data,ack) = DCE_info_in?data : S2(succ(send),data,ack)
S1(send<>nr,data,ack) = S2(send,data,ack)
S2(send,data,ack) = DCE_ack?ack: S3(send,data,ack)
S3(send,data,ack) = DCE_prt_out!data!send!ack : S4(send,data,ack)
S4(send,data,ack) = start: S0(send,data,ack)

timer DCE:
T0 = start: T1
T1 = start: T1 + tau: T2
T2 = timeout: T0 + start: T1

The receive_data process operates as follows: Wait for a frame from the channel_port. This frame consists of data, a frame number and a piggyback number (the number of the expected frame of the DTE). The piggyback will be sent to the send_data process, via DCE_send. Depending on whether the received frame is the expected frame or not, the data will respectively be sent to DCE_info_out or rejected. If the expected frame number was correct the exp variable will be increased. This is done by the succ function. In this case it is the same as inverting the exp bit, because exp can only be 0 or 1. Finally the expected frame number will be transmitted to the send_data process so it can be sent as a
piggyback with the next frame to be sent.

![Diagram of protocol processes](image)

**Figure 5. Scheme of the protocol 4 processes**

The send_data process operates as follows: Wait for an acknowledgement of a sent variable. If this takes too long, a timeout will be generated and the previous frame will be sent again. If an acknowledgement is received and this acknowledgement is not the acknowledgement of the last sent frame, the frame will also be resent, but with a new piggyback number. If the acknowledgement of the last frame has been received, new data will be read and a new frame, also with a new piggyback number, will be sent. We assume that there always is information to send (otherwise layer 3 could send an empty frame). The timer will be started after every sent frame.

The timer can be started and after a certain time (the length is not specified) the timer will generate a timeout. The timer could also be integrated in the channel as suggested in [KOO] chapter 4, but this has not been done, because later on in the LAPB protocol the timer is not only started after sending a frame.
4.2.2 The specification of the Channel
The channel is specified as two independent parallel processes with two states each, receiving a frame and sending it to the other side or losing it by a transmission error. The validation check, transferring into frames and appending flags, will be done in the channel. We assume that the channel will only transmit valid frames and reject every invalid frame.

Channel 1:
\[
\begin{align*}
\text{C11} & \quad = \quad \text{DCE}_\text{prt}\text{.out}\text{data}\text{send}\text{ack} \colon \text{C12} \\
\text{C12} & \quad = \quad \text{DTE}_\text{prt}\text{.in}\text{data}\text{send}\text{ack} \colon \text{C11} + \tau_\text{C11}
\end{align*}
\]

Channel 2:
\[
\begin{align*}
\text{C21} & \quad = \quad \text{DTE}_\text{prt}\text{.out}\text{data}\text{send}\text{ack} : \text{C22} \\
\text{C22} & \quad = \quad \text{DCE}_\text{prUn}\text{data}\text{send}\text{ack} : \text{C21} + \tau_\text{C21}
\end{align*}
\]

4.2.3 The specification of the DTE
The specification of the DTE is the same as the DCE, only it uses no timer. The CCS specification will be:

receive data DTE:
\[
\begin{align*}
\text{R0(\text{exp})} & \quad = \quad \text{DTE}_\text{prt}\text{.in}\text{data}\text{nr}\text{ack} \colon \text{R1(\text{exp, nr, ack})} \\
\text{R1(\text{exp, nr, ack})} & \quad = \quad \text{DTE}_\text{send}\text{ack} \colon \text{R2(\text{exp, nr})} \\
\text{R2(\text{exp=nr})} & \quad = \quad \text{DTE}_\text{info}\text{.out}\text{data} \colon \text{R3(\text{succ(\text{exp})})} \\
\text{R2(\text{exp<>nr})} & \quad = \quad \text{R3(\text{exp})} \\
\text{R3(\text{exp})} & \quad = \quad \text{DTE}_\text{ack}\text{!nr} \colon \text{R0(\text{exp})}
\end{align*}
\]

send data DTE:
\[
\begin{align*}
\text{S0(\text{send, data, ack})} & \quad = \quad \text{DTE}_\text{send}\text{!nr} \colon \text{S1(\text{send, data, ack, nr})} \\
\text{S1(\text{send=nr, data, ack})} & \quad = \quad \text{DTE}_\text{info}\text{.in}\text{data} \colon \text{S2(\text{succ(\text{send})}, data, ack}) \\
\text{S1(\text{send<>nr, data, ack})} & \quad = \quad \text{S2(\text{send, data, ack})} \\
\text{S2(\text{send, data, ack})} & \quad = \quad \text{DTE}_\text{ack}\text{ack} \colon \text{S3(\text{send, data, ack})} \\
\text{S3(\text{send, data, ack})} & \quad = \quad \text{DTE}_\text{prt}\text{.out}\text{data}\text{send}\text{ack} \colon \text{S0(\text{send, data, ack})}
\end{align*}
\]

4.2.4 Calculation of the parallel behaviour.
For this purpose a program called "CCS-tool" is used. This is a tool developed on the Eindhoven University of Technology which can calculate the parallel behaviour of CCS agents. This program can't treat variables. Thus, the specification is changed in the following one:

receive data DCE:
\[
\begin{align*}
\text{RC0} & \quad = \quad \text{DCE}_\text{prt}\text{.in} : \text{RC1} \\
\text{RC1} & \quad = \quad \text{DCE}_\text{send} : \text{RC2} \\
\text{RC2} & \quad = \quad \tau_\text{RC3} + \tau_\text{RC4} \quad (* \text{frame is valid or not} *) \\
\text{RC3} & \quad = \quad \text{DCE}_\text{info}\text{.out} : \text{RC4}
\end{align*}
\]
RC4 = DCE_ack: RC0

send data DCE:
SC0 = DCE_send: SC1 + timeout: SC4
SC1 = tau: SC2 + tau: SC3 (* frame resend or not *)
SC2 = DCE_info_in: SC3
SC3 = DCE_ack: SC4
SC4 = DCE_prt_out: SC5
SC5 = start: SC0

timer DCE:
T1 = start : T2
T2 = start : T2 + tau: T3
T3 = start : T2 + timeout : T1

Channel 1:
C11 = DCE_prt_out: C12
C12 = DCE_prt_in: C11 + tau: C11

Channel 2:
C21 = DTE_prt_out: C22
C22 = DCE_prt_in: C21 + tau: C21

receive data DTE:
RT0 = DTE_prt_in: RT1
RT1 = DTE_send: RT2
RT2 = tau: RT3 + tau: RT4
RT3 = DTE_info_out: RT4
RT4 = DTE_ack: RT0

send data DTE:
ST0 = DTE_send: ST1
ST1 = tau: ST2 + tau: ST3
ST2 = DTE_info_in: ST3
ST3 = DTE_ack: ST4
ST4 = DTE_prt_out : ST0

In this specification every condition is replaced by an internal action. This means that the specification is less specific, but it is now possible to calculate the parallel behaviour. If some stronger tools to calculate with specifications are developed in the future, it has to be possible to prove that the system is reducible to the first specification we made.

To show the method how to reduce the specification we give a reduction of several
parallel processes. As mentioned in [KOO] page 39, it is better to start with expanding that part of the system which communicates intensively. That explains why we will start with calculating the parallel behaviour of process timer and the process DCE_SEND.

send data DCE:

SC0 = DCE_send: SC1 + timeout: SC4
SC1 = tau: SC2 + tau: SC3 (* frame resend or not *)
SC2 = DCE_info_in: SC3
SC3 = DCE_ack: SC4
SC4 = DCE_prt_out: SC5
SC5 = start: SC0

timer DCE:

T1 = start: T2
T2 = start: T2 + tau: T3
T3 = start: T2 + timeout: T1

S = (SCIT) \ {start,timeout}

SC0T1 = DCE_send: SC1T1
SC1T1 = tau: SC2T1 + tau: SC3T1
SC2T1 = DCE_info_in: SC3T1
SC3T1 = DCE_ack: SC4T1
SC4T1 = DCE_prt_out: SC5T1
SC5T1 = tau: SC0T2
SC0T2 = DCE_send: SC1T2 + tau: SC0T3
SC1T2 = tau: SC2T2 + tau: SC3T2 + tau: SC1T3
SC2T2 = DCE_info_in: SC3T2 + tau: SC2T3
SC3T2 = DCE_ack: SC4T2 + tau: SC3T3
SC4T2 = DCE_prt_out: SC5T2 + tau: SC4T3
SC5T2 = tau: SC0T2
SC0T3 = DCE_send: SC1T3 + tau: SC4T1
SC1T3 = tau: SC2T3 + tau: SC3T3
SC2T3 = DCE_info_in: SC3T3
SC3T3 = DCE_ack: SC4T3
SC4T3 = DCE_prt_out: SC5T3
SC5T3 = tau: SC0T2

The next rewrite rules can be used:

18 -> 17 (tau reduction) SC4T3 = DCE_prt_out: SC0T2
17 -> 16 (SC4T3 = SC4T1) SC3T3 = DCE_ack: SC4T1
16 -> 15 (SC3T3 = SC3T1) SC2T3 = DCE_info_in: SC3T1
15 -> 14 (SC2T3 = SC2T1) SC1T3 = tau: SC2T1 + tau: SC3T1
14 -> 13 (SC1T3 = SC1T1) SC0T3 = DCE_send: SC1T1 + tau: SC4T1
12 -> 11 (tau reduction) SC4T2 = DCE_prt_out: SC0T2 + tau: SC4T1
This specification is now reduced to:

\[
\begin{align*}
\text{SC0T1} &= \text{DCE_send: SC1T1} \\
\text{SC1T1} &= \text{tau: SC2T1} + \text{tau: SC3T1} \\
\text{SC2T1} &= \text{DCE_info_in: SC3T1} \\
\text{SC3T1} &= \text{DCE_ack: SC4T1} \\
\text{SC4T1} &= \text{DCE_prt_out: SC0T2} \\
\text{SC0T2} &= \text{DCE_send: SC1T2} + \text{tau: SC0T3} \\
\text{SC1T2} &= \text{tau: SC2T2} + \text{tau: SC3T2} + \text{tau: SC1T1} \\
\text{SC2T2} &= \text{DCE_info_in: SC3T2} + \text{tau: DCE_info_in: SC3T1} \\
\text{SC3T2} &= \text{DCE_ack: SC4T2} + \text{tau: DCE_ack: SC4T1} \\
\text{SC4T2} &= \text{DCE_prt_out: SC0T2} + \text{tau: DCE_prt_out: SC0T2} \\
\text{SC0T3} &= \text{DCE_send: SC1T1} + \text{tau: DCE_prt_out: SC0T2}
\end{align*}
\]

and using the tau rules it can be reduced to:

\[
\begin{align*}
11 \rightarrow \text{tau-2} & \quad \text{SC4T2} = \text{tau: DCE_prt_out: SC0T2} \\
11 \rightarrow 10 & \quad \text{SC3T2} = \text{DCE_ack: tau: SC4T1} + \text{tau: DCE_ack: SC4T1} \\
10 \rightarrow \text{tau-1, tau-2} & \quad \text{SC3T2} = \text{tau: SC3T1}
\end{align*}
\]

This will lead to the following specification:

\[
\begin{align*}
\text{SC0T1} &= \text{DCE_send: SC1T1} \\
\text{SC1T1} &= \text{tau: SC2T1} + \text{tau: SC3T1} \\
\text{SC2T1} &= \text{DCE_info_in: SC3T1} \\
\text{SC3T1} &= \text{DCE_ack: SC4T1} \\
\text{SC4T1} &= \text{DCE_prt_out: SC0T2} \\
\text{SC0T2} &= \text{DCE_send: SC1T1} + \text{tau: SC4T1}
\end{align*}
\]

Because one process has to start with sending a frame we suggest that this process will start in state SC2T1, which means that the DCE will send the first frame. The result of this is that state SC0T1 is never reached, so we can eliminate this state.

The Calculation of the parallel process between the timer/sender process and the channel will lead to:

\[(S\{C1\} \setminus \{\text{DCE_prt_out}\})\]

Channel 1:

\[
\begin{align*}
\text{C0} &= \text{DCE_prt_out: C1} \\
\text{C1} &= \text{DTE_prt_in: C0} + \text{tau: C0}
\end{align*}
\]
send data DCE: (after relabelling)

\begin{align*}
S0 & = \text{DCE\_info\_in}: S1 \\
S1 & = \text{DCE\_ack}: S2 \\
S2 & = \text{DCE\_prt\_out}: S3 \\
S3 & = \text{DCE\_send}: S4 + \tau: S2 \\
S4 & = \tau: S0 + \tau: S1
\end{align*}

\[(SIC1)\setminus\{\text{DCE\_prt\_out}\}

\begin{align*}
S0C0 & = \text{DCE\_info\_in}: S1C0 \\
S1C0 & = \text{DCE\_ack}: S2C0 \\
S2C0 & = \tau: S3C1 \\
S3C1 & = \text{DTE\_prt\_in}: S3C0 + \tau: S3C0 + \text{DCE\_send}: S4C1 + \tau: S2C1 \\
S4C1 & = \text{DTE\_prt\_in}: S4C0 + \tau: S4C0 + \tau: S0C1 + \tau: S1C1 \\
S0C1 & = \text{DTE\_prt\_in}: S0C0 + \tau: S0C0 + \text{DCE\_info\_in}: S1C1 \\
S1C1 & = \text{DTE\_prt\_in}: S1C0 + \tau: S1C0 + \text{DCE\_ack}: S2C1 \\
S2C1 & = \text{DTE\_prt\_in}: S2C0 + \tau: S2C0 \\
S3C0 & = \text{DCE\_send}: S4C0 + \tau: S2C0 \\
S4C0 & = \tau: S0C0 + \tau: S1C0
\end{align*}

We know that the information sent first has to be received at the other side, before a new frame with an acknowledgement is sent back. So we can define the causal relation ([KOO] page 81):

\[\text{DCE\_prt\_in} \rightarrow \text{DTE\_send}.\]

This means that first the previous frame has to be received before the next frame is sent.

After this information the specification is changed in:

\begin{align*}
S0C0 & = \text{DCE\_info\_in}: S1C0 \\
S1C0 & = \text{DCE\_ack}: S2C0 \\
S2C0 & = \tau: S3C1 \\
S3C1 & = \text{DTE\_prt\_in}: S3C0 + \tau: S3C0 + \text{DCE\_send}: S4C1 + \tau: S2C1 \\
S2C1 & = \text{DTE\_prt\_in}: S2C0 + \tau: S2C0 \\
S3C01 & = \text{DCE\_send}: S4C0 + \tau: S2C0 \\
S3C02 & = \tau: S2C0 \\
S4C0 & = \tau: S0C0 + \tau: S1C0
\end{align*}

after reduction:

\begin{align*}
S0C0 & = \text{DCE\_info\_in}: S1C0 \\
S1C0 & = \text{DCE\_ack}: S3C1 \\
S3C1 & = \text{DTE\_prt\_in}: S3C01 + \tau: S3C1 + \tau: (\text{DTE\_prt\_in}:S3C1 + \tau: S3C1) \\
S3C01 & = \text{DCE\_send}: S4C0 + \tau: S3C1 \\
S4C0 & = \tau: S0C0 + \tau: S1C0
\end{align*}

The agent S3C1 is generating one or more DTE\_prt\_in actions. With using fairness [KOO
, chapter 4], we can reduce an agent like:

\[ A = B + \text{tau: } A \]

to

\[ A = B \]

After some tau moves, which means loss of the frame in the channel, a frame will be sent over in order to get the behaviour of B. If this would not be true, the channel would be useless, which we don't suppose. We can see the same effect in agent S3C1 in this specification, only in this case the action DTE_prt_in can be sent one or more times. We can simplify this specification to:

\[
\begin{align*}
S0 &= \text{DCE_info_in: } S1 \\
S1 &= \text{DCE_ack: } S2 \\
S2 &= \text{DTE_prt_in: } S3 + \text{tau: (DTE_prt_in: } S2 + \text{tau: } S2) \\
S3 &= \text{DCE_send: } S4 + \text{tau: } S2 \\
S4 &= \text{tau: } S0 + \text{tau: } S1
\end{align*}
\]

For the rest of the expansion the tool was used. After the expansion of \((\text{CiCS|CT|T}) \setminus \{\text{DTE_prt_out, DTE_prt_in, DCE_prt_out, DCE_prt_in}\}\) we get a machine with 96 States, but without deadlocks. The tool used is not powerful enough to reduce this scheme. It has to be possible to reduce the scheme and prove that it is (after tau reduction) equal to our starting point:

\[
\begin{align*}
S0 &= \text{DCE_info_in: } S1 + \text{DTE_info_in: } S2 \\
S1 &= \text{DTE_info_out: } S0 + \text{DTE_info_in: } S3 \\
S2 &= \text{DCE_info_in: } S3 + \text{DCE_info_out: } S0 \\
S3 &= \text{DTE_info_out: } S2 + \text{DCE_info_out: } S1
\end{align*}
\]

If it is possible to prove this, we know that if you implement your processes as defined above, it is a correct working communication protocol, without losing or doubling frames.

### 4.3 Translation into LOTOS

To show the relation between CCS and LOTOS, a translation of this protocol into LOTOS is made. In LOTOS it is necessary to specify the types used apart from the behaviour part. This specification starts with the declaration of data, a queue of octets. Afterwards the type frame number, which is 1 or 0 is defined. Hereafter the DCE behaviour part is
declared, consisting of 3 processes: send, receive and timer as displayed in figure 5.

```
protocol specification examples

library octet endlib

type data is octet
sorts data
opns create:  -> data
         add:  octet, data -> data
         first: data -> octet
         error:  -> octet

eqns forall d,e: octet, z:buffer
          ofsort octet
          first(create) = error;
          first(add(d,z)) = d;
endtype

library framenumber endlib

type framenumber
sorts fnum
opns 0,1:  -> fnum
       succ: fnum -> fnum

eqns ofsort fnum
       succ(0) = 1
       succ(1) = 0
endtype

behaviour DCE_protocol[DCE_info_in,DCE_info_out,DCE_prt_in,DCE_prt_out]

process DCE_protocol[DCE_info_in,DCE_info_out,DCE_prt_in,DCE_prt_out]:noexit:=
  hide send, ack, start, timeout in
  send[info_in, prt_out, send, ack, timeout, start](0,create,0)
  | |
  receive[info_out, prUn, send, ack](0)
  | |
  timer[timeout, start]
where
  process send[info_in, prt_out, send, ack, timeout, start]
            (send:fnum, data:data, ack:fnum):noexit:=
            (send?nr:fnum; ack?ack:fnum;
               [send eq nr] -> (info_in?data; prt_out!data!send!ack;
               send[info_in,prt_out,send,ack,timeout,start] (succ(send),data,ack)
               ]
               [send ne nr] -> prt_out!data!send!ack;
               send[info_in,prt_out,send,ack,timeout,start] (send,data,ack)
               ]
            timeout; prt_out!data!send!ack;
            send[info_in,prt_out,send,ack,timeout,start] (send,data,ack)
endproc
```
The specification in LOTOS is more complete and precise than the specification in CCS. The variables used are defined, while in CCS there is no possibility to specify variables. Furthermore, apart from defining the CCS like state machines it is possible to express the structure of the resources which are used.

4.4 The LOTOS specification of the LAPB protocol

In the previous paragraph a translation of a basic protocol into CCS and LOTOS is shown. In this chapter we will express a protocol named LAPB into the language LOTOS. The definition of the LAPB protocol is given in the X25 protocol CCITT recommendations [X25].

Protocol 4 is also a layer 2 protocol, but less intelligent than the LAPB protocol. In figure 6 a diagram is shown, which point out the different resources of the LAPB protocol. The host named DTE (Data Terminal Equipment) and the telephone company equipment named DCE (Data Circuit-terminating Equipment) are communicating with each other through an unreliable channel. The error corrections and recovery methods of the LAPB protocol will provide an error free communication between the two terminals.

The LAPB protocol is based on protocol 4, but it has some advanced methods to increase the efficiency of an unreliable layer 1 channel. For example it is possible to have more outstanding non-acknowledged frames (7 or 127 depending on the mode). Commands dealing with connecting and disconnecting are also present in this protocol.
In the next paragraphs, this X25 LAPB protocol will be specified in a resource oriented way. The decision to split the DTE in several parts, as you can see in the next chapter, is an advisable one, because making a constraint oriented specification of the whole DTE is very difficult. But after having some more experience with specifications, we found it better to specify the LAPB information transfer phase in a constrained oriented way, because it is more implementation independent. Nevertheless it is a good exercise to make the specification in a resource oriented way, as explained further on.

4.4.1 LAPB Commands
In table 1 a list of the commands used is shown. A command division in three groups can be made:
- Information frame: the frames containing the information which has to be transferred.
- Supervisory frame: the frames used for error correction by resending and for controlling transmission speed.
- Unnumbered frames: the frames for connecting and disconnecting or for the case of an unrecoverable error.

In figure 7 a partition of how you can build up the LAPB protocol is displayed. First the bit stream received from layer one has to be translated in frames mentioned in table 1. The process send bits deals with this by respectively cutting of the flags, checking the CRC check sum. Afterwards this process divides it into U frames (as specified in table 1) which are sent to the connection or disconnection processes, or into I and S frames which are sent to the data transfer phase or disconnected phase, depending on in which state the
Table 1. LAPB commands and responses

<table>
<thead>
<tr>
<th>Format</th>
<th>Command</th>
<th>Response</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information</td>
<td>I (information)</td>
<td>N(S), Poll, N(R)</td>
<td></td>
</tr>
<tr>
<td>Supervisory</td>
<td>RR (receive ready)</td>
<td>Poll/Final, N(R)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RNR (receive not ready)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REJ (reject)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unnumbered</td>
<td>SABM (set asynchronous balanced mode)</td>
<td>Poll</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DISC (disconnect)</td>
<td>DM (disconnect mode)</td>
<td>Final</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UA (unnumbered acknowledge)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FRMR (frame reject)</td>
<td></td>
</tr>
</tbody>
</table>

systems operate (disconnected or connected). The process Receive bits provides the same thing but visa versa. The process connection is responsible for the data link set-up, so this process will disable the process disconnected phase and will initiate the data transfer phase. After a connection made, process disconnect will be started. The process disconnection will take care of the disconnection during the data transfer phase. It will, when a disconnection demand has arrived, disable the data transfer phase and enable the disconnected phase.

In figure 7 the ports busy and error are also specified. Busy is used for indication of a full channel buffer, which is receiving the bits from the DCE. If this buffer is full, the DTE will send a RNR message. The port error is used for indication of an unrecoverable error in case of an idle channel state, N2 times a timeout or an impossible frame number.

4.4.2 Information transfer phase

To still keep the specification comprehensive we will start with only specifying the process data transfer phase. This process has been chosen, because it is the most complicated one, so the other processes will not supply many difficulties to specify. Figure 8 displays how the process data transfer phase is built up.

The information transfer phase is split into four parts. The process receive data is defined
to receive the frames and to send the information part to layer 3 and the supervision part like piggyback number, reject, acknowledge etcetera to the send data and send buffer processes which handle it. Send data will take care of sending information and supervision frames. Send buffer, which contains the buffer of unacknowledged sent frames, will handle resending of frames. It always offers the next frame to be send to the send data process. The process Timer will take care of time-outs. This timer is started if the process send data transmits a frame. It will be stopped after process send buffer will receive an acknowledgement of a non-acknowledged frame. If this is not the last sent frame, the timer will be restarted.

The specification of the process information transfer phase exists of 2 pages variable definition and 2 pages behaviour. It is displayed in appendix 3.
Using FDT LOTOS to derive tests

4.4.3 Type and Variable definition part
As usual, a LOTOS specification starts with a data specification part. Some standard types are existing in LOTOS [ISO3]. The standard types used here are: boolean, natural number and octet, as you can see in the second line of the specification.

The type data is defined as a queue of octets. This type is comprised of two constants create, which is an empty queue, and error to indicate an error situation. It also contains also two operators, first, the first element of the queue and add to fill the queue. The equations indicate that the first element of an empty list is not existing, so this will generate an error. If an element is added to the list, this will become the first element of the list.

The type ext_nat is a natural number with the extra operators minus, modula and between added. The operator between will check if variable x is an element of the buffer specified by the limits y and z. This operator is used to check if an acknowledgement in the cyclic buffer is allowed or not.

Figure 8. The structure of the Information Transfer Phase processes.
Buffer is the most important type. It is used to buffer the data in the process send buffer, until an acknowledgement is received. This buffer is a FIFO (First in first out) buffer. We use the same elements as used in the type data and some extra functions as remove, get_el, cut, length. Remove is used to remove the first element. When an acknowledgement arrives zero, one or more elements will be removed. For this case the function cut is defined. With the function get_el you can take an element out of the buffer to resend it. The function length will give the amount of elements that are in the buffer. This function is needed to indicate a full buffer.

The type definitions mode and frame_types are used to define some constants, which are used in the LAPB recommendations.

4.4.4 The LAPB information transfer behaviour part.

Process information transfer is the LOTOS version of figure 8. All the four processes are concatenated parallel, with their communication ports as synchronization. All the internal communication is hidden from the environment.

Process timer, which provides time-outs, has some special functions. If the timer is running and an other start request arrives, this action is excepted, but the timer doesn’t change anything, so also not its remaining time. The timer can be started over, by first stopping it and then restarting it. If the timer runs out N2 times there will be sent an error message and the data transfer phase will be disabled, by the disconnection block (not defined here).

The process send_buffer will take care of the buffer as defined in the type buffer. It provides four functions: Sending a buffer element to the send frame block, Buffering incoming data, handling an incoming acknowledge and handling an incoming resend request. The send buffer contains three variables: buf the buffer, Vs the framenumber of the last send element, el the buffer location of the last element.

For example every outstanding frame is acknowledged and the last sent frame has a number Vs = 3. The buffer will be empty and el will be 0. Some information is coming in, say 5 elements. As displayed in line two of this process specification each of these elements will be added to the buffer. Afterwards send_frame wants to send three frames, which will be frame 3, 4 and 5. Variable el and Vs are increased 3 times, so they will respectively be 6 and 3. Now a resend of frame 4 is coming in. The timer will be stopped, one buffer element will be cut of by the function cut, framenumber Vs will be 4 and el will be 0, what indicates that the buffer will be resend. Afterwards frame numbers 4,5,6,7 and 0 can be sent. If an acknowledgement of element 7 arrives, the timer will be stopped.
and restarted because there is an other outstanding frame left. Further the buffer will be
shrunk until one element is left, variable Vs will be unchanged and variable el will be 1
afterwards. This can be continued until doomsday.

Some conditions are added to avoid errors. In the first condition [el lt length] an element
can’t be send if the last buffer element is already send. In the second condition
[length(buf) lt k] incoming data can only be handled if the buffer is not full (if less then k
elements are present in the buffer). In the two conditions whereafter the process stops, the
process received an illegal frame number ns. In this case the process will send an error
message and will stop the process.

The process send_frame provides the sending of frames. The process consists of three
possibilities and one subprocess which can disable the process. After every confirmation of
a received frame the variable Vr will be changed. Sometimes (after an internal decision) a
supervisory frame will be send. If an information frame exist to be sent, this will be done
and the timer will be started. process send_s_frame will handle supervisory frame sending,
in case of a timeout, a poll, a busy condition and a reject condition. If the DCE is in a
busy condition, a loop in this process will provide that only supervisory frames are sent.
An other confirm function updates the sent piggyback frame.

The process rec_frame is responsible for the reception of incoming frames. After a
received I frame, it will send the piggyback number via ackn to the send_buffer process. It
will check if the framenumber of the received frame is the same as the expected number.
If so it will send the data to info_out and instruct the sending part to send a confirm, if
not, it will throw the data away and instruct the sending process to send a reject frame.
When a supervisor frame is received, an acknowledge, resend and/or wait message will be
transmitted to the send data block.

4.5 Specification of Connection Disconnection Processes

In the next chapter the specification of LAPB Connection Disconnection process is used to
derive a test suite automatically. The following specification is defined:

specification connection_disconnection[LI,LO,UI,UO]:noexit

library NaturalNumber, DecNatRepr, Boolean endlib

type verdict is Boolean
Protocol specification examples

sorts verdict

opns pass, fail : -> verdict
_eq_ : verdict, verdict -> Bool
eqns ofsort Bool
  pass eq pass = true;
  fail eq fail = true;
endtype

type L_Frame is
sorts L_Frame
opns SABM, UA, DISC, DM : -> L_Frame
endtype

type U_Frame is
sorts U_Frame
opns ConReq, DiscReq: -> U_Frame
endtype

behaviour disc_con[LI, LO, UI, UO]

where

process disc_con[LI, LO, UI, UO]: noexit :=
  (disconnection_phase[LI, LO, UI, UO] [> connection[LI, LO, UI, UO] >>
  accept result:: verdict in
    [result eq pass] -> (information_transfer_phase[LI, LO, UI, UO]
      [> disconnection[LI, LO, UI, UO] >> disc_con[LI, LO, UI, UO])
    []
    [result eq fail] -> disc_con[LI, LO, UI, UO]
where
process connect_res[LI, LO, UI, UO](tries:nat): exit(verdict) :=
  [tries ne NatNum(5)] -> ( LO! SABM;
    (LI! SABM; LO! UA; exit(pass)
      []
      LI! UA; exit(pass)
    []
    LI! DISC; LO! UA; exit(fail)
    []
    i; connect_res[LI, LO, UI, UO] (succ(tries)) )
Using FDT LOTOS to derive tests

This part of the specification is the LOTOS presentation of the 4 processes disconnection, connection, information transfer phase and disconnected phase. This specification is simplified, by eliminating the exit operators. In appendix 2, this specification is displayed. The behaviour of these two specification is similar, only the relation between the four processes as given above is removed. The number of states of this second specification is about 10 if it is translated into a finite state machine. In figure 9 this state machine is shown. The translation of this LOTOS specification into an extended finite state machine is done by hand. In chapter 6 a derivation by a tool is described which has one extra state, but the LOTOS EFSM specification also shown in appendix 2 and drawn in figure 9 is used as example in the next chapters.
Figure 9. The state machine of the Connection Disconnection LAPB protocol
5 Test suite derivation

5.1 Introduction

In the previous two chapters we saw how to describe a specification in the formal language LOTOS. The main purpose of my graduation project is to derive test suites from a formal specification. Several articles are written with this subject [BR1, BR2, DUB, GUE, HOG, NAI, PIT, TRE, TRI, WEZ]. Apart from the methods, also some tools are developed for this purpose. Most of these tools are still not available on the market. This chapter delineates the method which is preferred.

This chapter starts with a paragraph about what we want to test and how we do it. Afterwards the method used is introduced and the different phases of the method are elucidated in section 3. Finally the pros and cons of the method are pointed out.

5.2 The Testing Block

In figure 10 the testing model which is used is drawn up. The tester consists of an upper tester and a lower tester. The specification made is a specification of the implementation under test (IUT). The upper tester generates the communication which is normally done by the upper level. The lower tester is responsible for the communication of the lower level. These UT and LT are derived by the method, using the IUT specification as source. In case of testing LAPB, the upper level is the network layer and the lower layer is the physical layer.

The test itself is normally done by a test machine called CHAMELEON 32. This machine can be programmed in C. Some programs are developed which can translate a text presentation of a flow chart in the language C and the flow chart can be executed by the CHAMELEON machine.

Because several testing methods are used within the Philips Company it is preferable to generate a test suite in an independent language. After reading some articles about this subject, it was clear that the language TTCN was usable for this purpose. It is a common
used language for testing, the language is easy to translate in a flow chart and it will be standardized by the ISO in the near future. TTCN consists of a concatenation of transitions which are send or received from the IUT. If the test is completed one of the three verdicts fail, inconclusive or pass are added. Other facilities like variable definition, jumping to a label, etcetera are also possible. For an extensive explanation [IS1] is referred. The following example points out the way of writing a TTCN test trace.

Title: SABM collision

(* defined manually *)

U!ConReq
L?SABM
L!SABM
L?UA
U!DiscReq
L?DISC
L!UA
L?OTHERWISE
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
?EXPIRATION OF TIMER
(* initialization step *)
(* evaluation step *)
(* verification step *)
(* termination step *)
(* machine in wrong state *)
(* machine doesn’t return UA *)
(* UA didn’t arrive in time *)
(* initialization failed *)
(* """ *)
This is an example derived from the connect disconnect LAPB process. Every line corresponds with one test event. An input event corresponds with a question mark in the TTCN text. An output event is given by an exclamation mark. If the indentation of two events is the same, then these are alternatives. For example L?SABM and ?OTHERWISE are alternatives. Every input of the IUT is replaced by an output in the test trace. ?OTHER­WISE means if an other event is arriving then the events defined before and with the same indent level. ?EXPIRATION OF TIMER, means if a frame doesn’t arrive in time. The timer is started after every sent frame to the IUT.

5.3 The method used

To refresh your memory, in figure 11 a simplified life cycle is drawn up with the main questions which are answered in this and the next chapter. The LOTOS specification, written in DOC 190 has to be translated in TTCN test traces. All these traces together are called a test suite. The standard used for DOC 170 is the TTCN language as explained in the previous paragraph.

![Figure 11. Life Cycle for testing purpose.](image)

To find a method for the derivations some articles [BR1, BR2, DUB, GUE, HOG, NAI, PIT, TRE, TRI, WEZ] were scrutinized. The articles [NAI, GUE] inspired me the most.
They both suggest to translate a specification (LOTOS was used as specification language) into an Extended Finite State Machine (EFSM). From this EFSM test traces in TTCN were generated, which is our main goal. First the translation into an EFSM and afterwards the derivation itself is explained.

5.3.1 Translation into an EFSM.
We can translate any sort of LOTOS specification into a state oriented specification, which contains no hiding and no parallel processes. This can be done by effectuating the rewrite rules which are existing in LOTOS. In fact, the specification is expanded by removing operators such as parallel composition and disable, and reduce the specification to a tree of alternatives. The number of states will be maximal the product of the number of states within each parallel process. Such a specification, which is in fact our EFSM, contains the following information \( M \), which can be seen as a 7-tuple:

\[ M = \langle J, N, V, j_0, J_F, R, f_0 \rangle \]

with:
- \( J \) is set of states
- \( N \) is set of transitions
- \( V \) is set of data declarations (variables)
- \( j_0 \in J \) is initial state
- \( J_F \) a subset of \( J \) is set of final states
- \( R \) is set of rules
- \( f_0 \) is set of initial value assignments to the variables of \( V \)

And \( R \) the set of rules itself is a 5-tuple:

\[ r = \langle a, j, j', p, f \rangle \]

with:
- \( a \) = an action
- \( j \) = From state
- \( j' \) = To state
- \( p \) = precondition
- \( f \) = sequence of variable assignments

The following specification is a part of the disconnection-connection LAPB protocol,
written as a LOTOS EFSM:

SPECIFICATION connection[U,L]:noexit (*EFSM output *)

BEHAVIOUR S1 [U,L]

WHERE

PROCESS S1[U,L]: noexit :=
    L?DM; S2[U,L](0) []
    U?ConReq; S2[U,L](0) []
    L?SABM; S3[U,L] []
    L?DISC; S4[U,L]
ENDPROC

PROCESS S2 [U,L] (x:nat): noexit :=
    [x lt N2] -> L!SABM; S5[U,L](x)
ENDPROC

PROCESS S3[U,L](x:nat): noexit :=
    L!DM; S1[U,L] []
    L!UA; S7[U,L]
ENDPROC

etc.

This EFSM contains a set of states, which each correspond to one LOTOS processes. The transitions are the events and the calls to the next processes, like L!DM; S1[U,L] in state S3. The data declarations are the variables which are given to the states, like x of the type nat in state 2. The initial state is state S1, because the behaviour starts in state 1 as shown in the second line of the specification.

The set of rules is giving the rules of the state transactions. If a machine is in the state j and the precondition p is true, it can be transferred to state j' under transmission or reception of action a and activating the sequence of variable assignments defined in f.

Because the derivation method which is used can't manage variables, we have to make a restriction to this EFSM. We simplify the machine M to a quadruple:
\[ M_s = \langle J, N, j_0, R_s \rangle \]

with:
J is set of states
N is set of transitions
\( j_0 \in J \) is initial and final state
\( R_s \) is set of rules

And \( R_s \) the set of rules itself is a triple:
\[ r_s = \langle a, j, j' \rangle \]

with:
\( a \) = an action
\( j \) = From state
\( j' \) = To state

The restriction will in fact eliminate every variable in the state machine and every condition will be guarded true. In this stage we are independent of the LOTOS specification, so if it is possible to transform a specification of an other language into this structure, you can use this for deriving a test suite.

5.3.2 Derivation of the test suite
The method used generates a test suite which contains one test trace for each input event of the EFSM. The name of the test trace will be the state where in the event starts, the state where in the event ends and the event itself. Such a test trace is split into four main parts:

- initialization step
- evaluation step
- verification step
- termination step

Initialization step:
This step constrains the path which brings the IUT from the initial state of the machine into the initial state of the event to be tested. For example in our connection disconnection machine, if we want to test "U?DiscReq" from state S7. We can use for example the next
paths:

L!SABM
  L?UA

or

L!DM
  L?SABM
  L!UA

The criterium used for selecting is searching for the path with the minimum amount of test events. In this case it will be the first one. Furthermore the inconclusive and failure paths will be added. If for example an internal decision is made to follow an other legal path, but without testing the intended event, the verdict of this path will be inconclusive. If a path like this exists it will be terminated as soon as possible. The termination path is explained later on.

To detect implementation errors the two paths "?OTHERWISE" and "?EXPIRATION OF TIMER" are added to every frame reception tree. It will also be preferable to attach some comment to notice where the bug in the IUT is to be found.

Finally the initial path for arriving in state S7 is:

L!SABM
  L?UA
    + next step (evaluation step)
  L?DM
    + termination step  inconclusive
 ?OTHERWISE  fail
 ?EXPIRATION OF TIMER  fail

Evaluation step:
This is the body of a test case. It sends the event named sev which we want to test. If any response frame is sent then it is accepted. If this event named rev is received the verification path of the state which is arrived is added. If no event is send back by the IUT and there is no exit to a next phase, the verification path of the present state is added to the test case after the timer starts and stops. This timer is used to make sure that nothing is arrived. If an internal action is possible after sending event sev then the same algorithm
has to be used for the state we are arriving in now. An "?OTHERWISE" with a verdict fail is added at any point where something different from what is expected to be received can occur. The evaluation step is summarized in the following algorithm:

\[
P!fs \quad \text{(* send the event to be tested to the IUT via an arbitrary port P *)}
\]

\[
\text{rec if reception} \quad \text{(* frame is excepted and there will be a reaction frame *)}
\]
\[
\begin{align*}
\{ \\
\quad & P'?fr \\
\quad & + \text{verification path of the new state}
\}
\end{align*}
\]

\[
\text{else} \quad \text{(* no reception *)}
\]
\[
\begin{align*}
\text{If no exit to a next phase} \\
\{ \\
\quad & \text{start timer} \\
\quad & ?\text{timeout} \\
\quad & + \text{verification path of this phase} \\
\quad & \text{?otherwise: fail}
\}
\end{align*}
\]
\[
\text{If an exit to a next phase} \\
\{ \\
\quad & \text{goto rec with new state} \\
\}
\]
\[
\text{?otherwise fail}
\]
\[
?\text{expiration of timer} \quad \text{fail}
\]

If a time out has to be tested within the IUT (as in our example from state S4 to state S2) the line "P!fs" is replaced by a timer with the minimum duration of the timer within IUT and the second timer will have a duration of maximum minus the minimum elapsed time. For every output and time out event there can be described an evaluation step in this fashion.

In our example we are evaluating "U?DiscReq". Afterwards reception of this event the IUT will send an output event "L!DISC". So our evaluation step will be:

\[
U!\text{DiscReq} \quad \text{pass}
\]
\[
L?\text{DISC} \quad \text{fail}
\]
\[
?\text{OTHERWISE}
\]
Verification step:
In this step we have to find a verification path which is unique for the state which is verified. This method is used to make sure that the IUT is arrived in the right state. To every event a variable is added which indicates if an event is unique, half unique or not unique. If an event is unique than no other equal event is used in the specification. Half unique means that the name is not unique but there is no other equal event with the same destination. In case of a not unique event, more equal events with the same destination occur in the IUT.

To find a unique path, we have to search for a unique event which is starting in the state to verify. This event can be an output event of the IUT or an input event of the IUT, which generates an output afterwards. If it is a unique event which ends up in a stable state, you didn’t deliver a proof, because no reaction event from the IUT is given. If it is not possible to directly find a path like this, you can move to an other state via a unique or half-unique event and verify this state. It is not guaranteed there will always be a unique path, but in practice a unique path can be almost always found.

The states of the IUT which are sending a unique event can be given the label "unique". In our example the states S2 and S9 have this label "unique". Also the IUT states which recept a unique event and will send an event afterwards will be given the label "unique". In this case the states S1, S7 are acquiring this label. So to test a state we have to find a unique or half unique event, or a path of only unique and half unique events to one of these unique states. On this manner we can create a hierarchy of state verification. For our example this will be:

Directly verifiable: S1, S2, S7, S9
After one event: S5, S10
(every half unique or unique event which end in state S1, S2, S7 or S9)

We only verify stable states, because the evaluation step will always end in a stable state. Thus in our example all the stable states can be verified.

S5 is verifiable by a path of two events. The problem is that it is done via an internal event (which is a timeout) and you don’t know the lack of time, so it is not preferable, but in this case it is the only possibility to verify.

In our example we have to check state 10. It has 3 different outputs one half unique and
two non unique. So we choose "L?UA" and we will verify state S1. State S1 has a unique event namely "U?ConReq" and the machine produces also the unique response "L!SABM". So the Verification step will be:

L!UA
  U!Conreq
  L?SABM
    + termination path
    L?OTHERWISE
    ?EXPIRATION OF TIMER

termination step:
This phase will bring the automate in the initial state, so you can chain the different test sequences. The same selection criterium as defined within the initialization step is used. The minimum amount of events is searched. In principle, it is better to generate test traces which are not only concatenated in the initial state, but also in other stable states. This means that the test suite, containing all the test traces, will become smaller. It will not be very difficult to change this strategy, but because my first intention was deriving a test suite, optimization was not yet considered.

In our example we have two termination paths, one from state S5 and one from state S1. The state S1 is the initial state so it doesn’t contain test events. The termination path of state S5 will be the following:

L!UA
  L!DM

So the whole obtained TTCN test trace example is:

title : (S7) - U?DiscReq -> (S9)

L!SABM
  L?UA
    U!DiscReq
    L?DISC
    L!UA
      U!Conreq
      L?SABM
For seeing more examples of test traces derived from this EFSM appendix 2 is referred to.

5.4 The pros and cons of this method

This method is chosen, because the algorithms used are relatively easy to implement in a C program as shown in the next chapter. Further on a tool is found which can generate an EFSM and also can simulate the protocol. In this way it is possible to use this tool to check the specification made and to derive the test suite. The method used is semi formal, because you can never proof that there will be a unique verification path, but in practice these are almost always found.

On this moment other more formal methods and belonging tools are developed, but the problem is still too difficult to apply this in practice. The method used here is still one of the most effective possibilities to generate tests.

Maybe in the near future the tool set LITE, which also consists SMILE will be expanded with a test generator.
6 Tools used and developed

6.1 Introduction

To follow the life cycle as defined in figure 11 two tools are used. One is available on the market and is called SMILE. This tool is developed on the University of Twente. Unfortunately this tool was not available until 3 days before the end of my graduation period. First an extended finite state machine which is normally derived by SMILE is derived by hand. The last 3 days an expansion with the tool SMILE was made, so the results could be compared.

The second tool, named 'Test Suite Generator' is developed by myself. This tool is an implementation of the theoretic derivation explained in Chapter 5. This tool is programmed in C and is executable on a SUN Unix system, the system which is also used for the SMILE tool.

This chapter contains a short evaluation of the tool SMILE and an explanation of how the tool 'Test Suite Generator' is built up.

6.2 SMILE transformation

The tool SMILE is a part of the toolset LITE[BOO]. This toolset LITE is available within the firm ITA (Information Technology Architecture B.V), Institutenweg 1 in Enschede, The Netherlands. At this moment this toolset consists of a syntax checker, a semantics checker, the simulator SMILE and the structure editor CRIE.

To test if this tool is capable to generate an EFSM, the specification of the LAPB connection, disconnection specified in Chapter 4 is used. The syntactics and the semantics checker were very useful to detect errors, only the error message were not very clear and specific. To get a better idea about this tool [LAU] is referred. This is an evaluation report of this toolset, which is made within the Philips Research Laboratory Eindhoven.

The specification written in appendix 2 is expanded and received 13 states. The difference
between this specification and the specification made by hand is in this case not very big. Only the tool SMILE doesn't optimize the specification. In this generated specification two transitions (State 8 and State 12) are created with the same functional behaviour. It is not very difficult to optimize this effect afterwards, but the question is if this tool is using rewrite rules to optimize.

To detect if the Tool smile is able to optimize, an other specification with a lot of exits, showed in appendix 4, is expanded. This specification is exactly the same specification as the one before, but the SMILE tool generates 36 states in stead of 13 (appendix 4).

To make the use of SMILE sufficient it is preferable to make your specification as short as possible and to avoid the operator exit. It will also be possible to design a tool which can optimize the SMILE output file, or maybe in future the tool SMILE will be expanded with a reduction tool.

Within this tool a place is also reserved for a test generator named COOPER. Because the tool is still very new this test generator is not available yet. My conclusion is that in spite of the lack of a reduction tool, this tool is very useful to generate the EFSM. If some extra internal actions are created, this doesn’t mean that the 'Test suite generator' is creating a test suite which is longer than without internal actions. Unfortunately it was not possible to test this.

6.3 Test suite generator

The theory explained in the previous chapter, about how a test suite can be derived, is implemented in a tool. The tool is called 'Test Suite Generator' and is written in language C. The sun work stations working under UNIX, which were available within TRT are used for writing the C text. This system is also used because the toolset LITE can work under this system, so an integration between these two tools can be made.

The 'Test Suite Generator' is partitioned into four modules. One is used for reading in the generated output file of the tool SMILE. In this module the information like unique events and shortest initial and termination paths are added. The SMILE EFSM will be placed in a C pointer structure, which is shown in the next paragraph. This C pointer structure can be used by the successive module called 'Generate test suite'. This module produces also a C pointer structure which contains a TTCN test suite, build up of several test traces (corresponding to the amount of input events in the EFSM specification). To represent the results of this two modules, the module output will make an ASCII file of the two
generated C pointer structures. One containing the EFSM with the added information and one containing the TTCN test trace (See appendix 2 for an example). The module main is written to call the functions developed in the other three modules.

The next paragraph gives an explanation of the used type definitions. Hereafter the different modules are described. The chapter will be finished with the results of the program usage and recommendations to improve the quality of the program.

6.4 Used type definitions

In figure 12 a design of the C pointer structure is displayed. The structure is divided into two main parts, states of the type "ext_state_type" shown in the second row and events of the type "event_type" shown in the third row.

A state consist of 6 variables:
- a pointer to the name, ports to communicate through and variables used.
- The state number.
- The state sort: stable or instable.
- a pointer to the state in succession.
- a pointer to the first event which starts in this state.
- a pointer to the first event which ends in this state.

An event consists of 10 variables:
- a pointer to the state where the event starts.
- a pointer to the state where the event ends.
- event action
- condition under which the action can occur.
- event sort: input, output, internal action or synchronization
- uniqueness of the event:
  - unique, half-unique or not unique
- event number
- a pointer to the next event which starts in the same state as this event.
- a pointer to the next event which ends in the same state as this event.
- a pointer to name, change of variable contents and change of ports (relabelling) of the state where this event is ending in.

To describe this variables in a C structure 4 types are defined. The first is the extended state type:
C Pointer Structure

```
struct ext_state_type {
    state_type *state;
    int num;
    stable_type sort;        /* state = stable or instable */
    ext_state_type *next;
    event_type *first;       /* first connected event to next state */
    event_type *first_from;  /* first event from which you can arrive at this state */
}
```

A variable of this type conforms to the state defined in figure 12. The variables first and
first_from are respectively used for detecting the possible events starting from this state and the events which point to this state. After reading in the EFSM, the variables first (first_from) are pointed such that the event given by this pointers is the first step of the shortest termination (initialization) path. This event itself has its "to state pointer" ("from state pointer"), which leads to the next state. If this is not the initial state than the first (first_from) pointer will be followed again to find this initial state.

The events are covered by the next type definition:

```c
struct event_type
{
    struct state_type *to;
    char event[], cond[];
    sort_type sort;
    int num;
    yes_no unique;
    event_type *next, *next_from;
    ext_state_type *fp,*tp;
}
```

In this definition fp and tp are used as the from_state_pointer and to_state_pointer. The type sort_type is an enumeration of the four constants: input, output, internal or syn(chronization). These are the four types of events which are used within the LOTOS EFSM. The type yes_no, declares apart from yes and no also half, for half unique events.

The type state_type contains the state name, the port names and variables corresponding to this state. This type is also used for a process call after a state. The field port names are used to define relabelling and the field variables are used to change the value of these variables.

```c
struct state_type
{
    char name[],port[],var[];
}
```

The length of the strings defined above are all 80, because normally names, port definitions, conditions, actions and variables are not containing more information than one line.

Finally the type which can contain the state machine is defined by:
struct efsm_type
{
    state_type *name,*first_state;
    ext_state_type *first;
}

Within the variable name, the name of the specification is defined. Within the variable first_state the sentence following the LOTOS operator behaviour is read in, to define the first state and its initial variable assignments. The pointer first_state is pointing to this first state.

These types together will be able to represent the EFSM C-structure. A variable of efsm_type contains all information defined in the 7-tuple M, which is discussed in the previous chapter.

The next defined type structures are necessary to describe the test cases in TTCN:

struct ttcn_el_type
{
    char cont[40], label[5], comment[20];
    verdict_type verdict;
    struct event_type *ev;
    struct ttcn_el_type *nl, *np; /* next level, next possibility */
};

struct ttcn_path_type
{
    char name[];
    struct ttcn_el_type *fel; /* first element */
    struct ttcn_path_type *next;
};

Every test trace covered by ttcn_path_type receives the name given by a character string of the event which is tested and its corresponding from and to state. The pointer fel points to the first TTCN element. The element next points at the next test trace which is present in the test suite.

Every TTCN elements contains eventually a cont(ence) written in text, like for example
"?OTHERWISE", a label to jump in the TTCN test trace and comment. If this element is a final element of the test trace it contains one of the verdicts: fail, inconclusive or pass. Ev points at the event which will be tested by this element. the pointer nl (next level) points at the element to be tested after this element is tested. the pointer np (next possibility) is pointing to the next alternative element which also can occur, so these are the elements at the same indent level of the TTCN test trace.

Two other types state_list_type and event_list_type are declared temporary for helping to search the test traces.

6.5 Module: Read EFSM

To read in the EFSM which is created by the tool SMILE, the module read EFSM is written. The module is called "efsmproc.c". This module consists of a main function called "read_smile" and twenty other functions. The function "read_smile" calls the following functions:

- read_efsm, to read in the file.
- count_states, to count the number of states.
- add_inverse_path, which fills the variables first_from and next_from.
- find_init_step is exchanging the "first_from" events with the events which follow the shortest initialization path.
- find_term_step is exchanging the first events on the same manner.
- search_unique_events adds to every event if it is not unique, half unique or unique.

The function read_efsm itself calls six subfunctions:

- str_form, to change a string into capitals and eliminate every unknown or unused character.
- action_sort, to detect the sort (input, output, internal or synchronization) of an event.
- search_state: After reading in the name of the next state after an event, the pointers to this state are searched by this function.
- scan_state, to read the name, ports and the variables of a state.
- scan_event, to read the condition, action itself, and scans the "to_state".
- read_state reads the state and all the events starting in this state. This corresponds with reading one LOTOS process.

To find the initialization and termination steps, the following function are added to this
Using FDT LOTOS to derive tests

module:

- add_list, which adds a state to a state list, if it is not already in the state list. It returns true if this element was not present.
- full_list will return true if the state list contains every state.
- swap_from_events and swap_to_events, will exchange the event which points to the shortest path and the first(from) event.

The function find_init_step will begin with a state_list containing only the initial state. Every event starting in this state is pointing to a new state. If this new state is not already in the state list this state is added to the state list and the first_from event of this state will be exchanged with the event which is followed to arrive in this state. After every event of the initial state is checked the algorithm will continue with the next state which is in the list. If the list contains every state the process will be terminated. The same algorithm is used for the termination step.

Finally the functions add_non_unique_event and add_half_unique_event helps the function search_unique_events to add the label unique. Every event is initialized as unique. The function add_half_unique_event will make a list of events with different names. If an event is found which has the same name as an event in the list, both events will get the label half_unique. In the same way the non unique events are found, only this function will make a list of events with different names and destinations (to_states).

The functions empty_event_list and empty_state_list are used to free allocated memory space which was used for finding the initialization and termination steps and the unique events.

6.6 Module: Generate test suite

The module "Generate_test_suite" called "TTCNGEN.C" in appendix 5 contains nine functions, with the main function gen_test_suite. This module makes extensive use of recursion. First the different functions are introduced and afterwards the algorithms used are explained. The following functions are defined:

- create_empty_element, which creates an element which can contain one test event. It initializes the variables.
- create_init_path, which has the function to build up the path from the initial state through to the state where the event to evaluate starts, given by st.
- create_term_path, which builds up the path from any state to the initial state.
- create_side_path, which creates the path which we don't want to test, but is also feasible to follow. It will be given the verdict which is presented by variable "verdict".
- create_verification_path, which searches a unique path for verification of a state.
- create_reception_path, which receives the events send by the IUT after the evaluation.
- create_eval_path, which will send the event to evaluate to the IUT and calls the reception path if a reaction event of the IUT is expected.
- create_path provides the concatenation of the different steps created in the other functions.
- gen_test_suite provides the chaining of all test traces to one test suite.

The functions which are searching the paths are making extensive use of recursion. This is done because during building up the different trees a lot of similar algorithms are used. If this algorithm is the same, the same function is called within a function, which leads to recursion.

The function create_path first calls the function create_init_path. This function returns with a pointer to the first element of the initialization step. If any elements are present in this step, the function will call itself (recursive) till the first element is found. This first element will be filled with the necessary information and the function will return with the pointer to this first element. If more elements are present in this path these are added to the tail of the queue, after every return action in the nested function.

This created path is expanded by side paths, due to the fact that an internal or alternative output action could occur. This is done by the function create_side_path. If an alternative path is found, it will be terminated with a termination path and will get the verdict inconclusive. Besides these paths the elements "?OTHERWISE" and "?EXPIRATION OF TIMER" are appended.

After the initial and side paths are found, the function create_eval_path is called. This function will return with a pointer to the first element of the evaluation path and every path which follows after this step. This means that this function itself calls the functions which are creating the rest of the steps. The first element is the event which we want to test. Beside this element the side_paths are added by function call to create_side_path. If the state which we are arriving in after this action is unstable we will continue the derivation with the function create_reception_path. Otherwise a timer event will be added to control if this state is stable or not. For the fact that an event is arriving before a time out an event "?OTHERWISE" with a fail verdict will be appended.
The reception path receives every possible event which is send by the IUT after sending the event to evaluate. If a reception event is found the verification path of the state arrived is added with a verification step by calling the function "create_verification_path". The function will also append the "?OTHERWISE" and "?EXPIRATION OF TIMER" elements.

The function create_verification_path searches for an adjacent event which is unique. If such an event is not found, the first half unique event is followed and the new arrived state is verified. The other possibilities are searched by the function create_side_path. After a unique path is found the function create_term_path is called. The function which creates the verification path doesn't completely conform with the algorithm defined in the previous chapter. This function has to be improved.

The function create_term_path will follow the "first events" to arrive in the initial state. In fact it is following the "first event" and calling itself (recursion).

The derivation method has some difficult structures, because it is needed that almost every step can be added in any stage of the TTCN test case. The most powerful way to create a path like this is using recursion. The disadvantage of this method is that it is difficult to find errors and the chance exists that an endless loop will be created.

6.7 Module: Output

This module called "output.c" is created for providing output to the screen or to a file. It consists of 6 functions:
- write_state, write_event, write_efsm for creation of the EFSM output file with the added information by the module read_efsm.
- write_test_event, write_elements and write_ttcn_path for writing the TTCN test suite which is created by the module gen_test_suite.

The output file will consist of a test suite, which is divided and will contain as many test cases as input events starting from a stable state. Every test case will get the reference "<state name>, <event> -> <next state>" which is tested. Afterwards the Behavioral description will show the events from the first state until the first state.

Every information which is included in the two types of C structures are translated in an ASCII file. An example of this file can be found in appendix 2.
Most functions are not difficult to understand. The only more complex function is the function write_elements which contains recursion. It will first print the information in the tcn_element and if a next element exists it will be printed before the alternative elements.

6.8 Results

The tool SMILE is able to generate an EFSM out of a LOTOS specification. This specification can also be checked on syntactics and semantics by the toolset LITE. The tool smile doesn’t optimize an EFSM, so this has to be done by an other tool, or maybe in newer versions of SMILE this option is included.

Unfortunately it was not possible to use the tool SMILE before testing the tool, so an example of the connection disconnection part of the LAPB protocol was written as an EFSM. This EFSM is the only one which is tested for deriving a test suite. If this program will be used in practice, it has to be tested more extensively and some algorithms can be improved, as suggested in this chapter.
7 Conclusions

The Formal Description Technique (FDT) LOTOS which is used to specify the example of the X25 LAPB protocol is a useful starting point for specification. Especially because LOTOS supplies the possibilities to calculate with a specification. In principle the language can be used in any part of the life cycle. A constraint oriented specification made in the specification phase of the life cycle can be translated into a resource oriented specification (implementation) in which the implementation decisions are already made. In the future it could be possible to calculate if this implementation conforms with the specification made in the specification phase.

With the tool SMILE and the "test suite generator" it is possible to derive test suites without bothering about variable handling. The TSG has only been tested for a small EFSM so this tool has to be tested more thoroughly. Also some algorithms used are not optimal and can be improved.

The method used doesn't need other restriction as mentioned above. Only selecting usable test traces from a long test suite could be difficult. Also the generation of an EFSM by the tool SMILE has not been optimized, so it is important to reduce this EFSM. To test the TSG it is recommendable to specify a protocol in a constraint oriented way and generate a test suite of one or a combination of several constraints. In this way we can check the generated test suite without having an immense test suite.

During my training period, it was still difficult to purchase LOTOS tools, but nowadays the toolset LITE is available. Within this toolset some bugs are still present, but the results were satisfying. In future a test generator will be integrated within this toolset. Unfortunately I could only use this toolset for two days.
Reference List

[ALD] Rudie Alderen
COOPER the compositional construction of a canonical tester,
Formal Description Techniques II
S.T. Vuong (Editor) Elsevier Science Publishers (North-Holland), 1990

[BER] Gilles Bernot, Marie Claude Gaudel and Bruno Marre
Software testing based on formal specifications: a theory and a tool,
Software Engineering Journal, November 1991

[BOL] Tommaso Bolognesi, Ed Brinksma
Introduction to the ISO Specification Language LOTOS,
Elsevier Science Publishers B.V. (North-Holland), 1987

[BOO] R.E. Booth, K.J. Turner and V.M. Jones
Formal methods for communication systems development
the ESPRIT project LOTOSPHERE
BT Technol J. Vol 10 No 2 April 1992

[BR1] Ed Brinksma, Rudie Alderen, Rom Langerakn Jeroen van de Lagemaat, Jan Termans
A Formal Approach to Conformance Testing,
Formal Description Techniques II
S.T. Vuong (Editor), Elsevier Science Publishers (North-Holland), 1990

[BR2] Ed Brinksma
A theory for the derivation of tests,
Protocol Specification Testing and Verification VIII,
Elseviers Science Publishers B.V. (North-Holland), IFIP, 1988

[BR3] Ed Brinksma
A tutorial on LOTOS,
Protocol Specification, Testing and Verification, V
M. Diaz (editor), Elsevier Science Publishers B.V. (North-Holland), IFIP 1986

Executing LOTOS Specifications,
Protocol Specification, Testing and Verification, VI
Elseviers Science Publishers B.V. (North-Holland), IFIP, 1987

[CLA] Robert G. Clark and Valerie M. Jones
Use of LOTOS in the formal development of an OSI protocol,
computer communications, vol 15 no 2 march 1992

[DIA] Michel Diaz
Conception formelle des protocoles et des services dans les systèmes distribués,
Génie Logiciel & Systèmes experts No 26, Mars 1992
Using FDT LOTOS to derive tests

[DRA] Lynne Drayton, Amanda Chetwynd, Gordon Blair
Introduction to LOTOS through a worked example, computer communications, vol 15 no 2 march 1992

[DUB] M. Duboc, G. v. Bochmann, O. Bellal and F. Saba
Translation from TTCN to LOTOS and validation of test cases, Formal Description Techniques III
J. Quemada, J. Mañas, E. Vázquez (Editors)
Elseviers Scince Publishers B.V. (North-Holland), 1991

[ELI] Peter van Eijk
Tools for LOTOS Specification Style Transformation, Formal Description Techniques II
S.T. Vuong (Editor), Elsevier Science Publishers (North-Holland), 1990

[GUE] Djaaffar Gueraichi and Luigi Logrippo
Derivation of Test Cases for LAPB from a LOTOS Specification, Formal Description Techniques II
S.T. Vuong (Editor), Elsevier Science Publishers (North-Holland), 1990

[GUI] Ph. Guillet
Manuel de methodologie dossier du logiciel, Procedure generale, FBW 105 225, TRT internal report

[HOG] Prof. Dr. Dieter Hogrefe
Conformance testing based on formal methods, Formal Description Techniques III
J. Quemada, J. Mañas, E. Vázquez (Editors)
Elseviers Scince Publishers B.V. (North-Holland), 1991


[IS2] ISO/DP9646-1.2 "OSI Conformance testing methodology and framework, part 1 & 2, general concepts and abstract test suite specification"


[JAG] Paul de Jager, Willem Jonker, Albert Wammes and Johan Wester
An Interactive programming environment for LOTOS, Formal Description Techniques III
J. Quemada, J. Mañas, E. Vázquez (Editors)
Elseviers Scince Publishers B.V. (North-Holland)

[KOO] C.J. Koomen

[LAU] Y.M. Lau
Evaluation of Lite toolset
Philips Natlab internal report, 1992, Eindhoven
Reference List

[LIN] Fuchun Joseph Lin, Ming T. Liu
Protocol Validation for Large-Scale Applications,
IEEE Software, January 1992

[LOG] L.Logrippo
The University of Ottawa LOTOS Toolkit,
Formal Description Techniques III
J. Quemada, J.Mañas, E. Vázquez (Editors)

[MIL] R. Milner

[NAI] Kshirasagar Naik, Behcet Sarikaya
Testing Communication Protocols,
IEEE Software, January 1992

[PAV] Santiago Pavón, Martin Llamas
The Testing Functionalities of LOLA,
Formal Description Techniques III
J. Quemada, J.Mañas, E. Vázquez (Editors)

[PEC] Charles Pecheur
Using LOTOS for specifying the CHORUS distributed operating system kernel,
computer communications, vol 15 no 2 march 1992

[PIT] David H. Pitt and David Freestone
The Derivation of Conformance Tests from LOTOS Specifications,

[TAN] Tanenbaum
Computer networks

[TRE] Jan Tretmans
Test Case Derivation from LOTOS Specifications,
Formal Description Techniques II
S.T. Vuong (Editor), Elsevier Science Publishers (North-Holland), 1990

[TRI] Piyu Tripathy and Behcet Sarikaya
Test Generation from LOTOS Specifications,

"Formal Methods to Assist Conformance Testing" A case study,
Formal Description Techniques III
J. Quemada, J.Mañas, E. Vázquez (Editors)

[WIN] Adam C. Winstanley and David W. Bustard
EXPOSE: an animation tool for process-oriented specifications,
Software Engineering Journal, November 1991

Using FDT LOTOS to derive tests

Appendices

by

Martijn A.M. ten Hacken


in charge of C.J. Koomen
Related to

Eindhoven University of Technology
Faculty of Electrical Engineering
Department Digital Systems

Den Dolech 2
5600 BM Eindhoven
The Netherlands
+31.40.479111

under daily supervision of Ph. Guillet
Related to

TRT
(Télécommunications Radioélectriques et Téléphoniques)

5, Avenue Réaumur - B.P. 21
92352 Le Plessis Robinson Cedex
France
+33.1.40.94.70.00
Contents

Appendix 1 Tutorial of the Test Suite Generator ........................................... 71
Appendix 2 An example of a LOTOS specification, its EFSM and the derived Test Suite .................................................. 73
Appendix 3 Specification of the LAPB protocol in LOTOS. ............................. 87
Appendix 4 An other example of a disconnection connection Specification, with the generated EFSM. ........................................ 91
Appendix 5 Source Text of the program Test Suite Generator ........................ 99
Appendix 1 Tutorial of the Test Suite Generator

Test Suite Generator
====================

This tool is designed for generating test suites from an extended finite state machine as created by the tool SMILE. The program consists of four modules namely:

efsmproc.c: a module for reading the input file in a c structure.
ttcngen.c: generation of the TTCN test suite from the c structure
output.c: providing the output of the TTCN test suite and the c structure.
main.c: main program

To run this program an executable file is created which can be started in the following way:

generate <inputfile> [outputfile]

inputfile: any file with the same syntax as a SMILE output file.
outputfile: will contain the efsm in the c structure and the TTCN test suite.

The c structure contains the next information:

specification: name, first_state, number of states
state: number, name, ports, variables, stable or instable
event: number, action, uniqueness(un, hu, nu),
        sort(input, output, internal, syn), condition, from state, to state

The TTCN test suite contains the following information:

For every input event(IE) of the implementation under test (IUT), which
 corresponds with the SMILE input file, a TTCN test trace will be created.
This TTCN trace is divided in 4 parts:
- initialization path: to arrive to the from state of the IE.
- evaluation path: effectuate the IE and receive the response (if any).
- verification path: search a unique path to check if the IUT arrived in the right state. If such a path can’t be found, a test case without the verdict pass will be generated.
- termination path: to set the IUT in the initial state so the test traces can be concatenated to one test suite.

TTCN distinguishes three sorts of verdict: pass, inconclusive and fail. Inconclusive means that there is no fault detected, but the test is not or partly done.

The tool contains the following files:

source: efsmproc.c, main.c, output.c, ttcngen.c, typedef.h
object: efsmproc.o, main.o, output.o, ttcngen.o
execution: generate
example: connect.lot
tutorial: tutorial.txt

12 files.
Appendix 2 An example of a LOTOS specification, its EFSM and the derived Test Suite

This specification is using four ports LI, LO, UI, UO which stands for respectively Lower tester Input, Lower tester Output, Upper tester Input and Upper tester Output. From this specification an EFSM is generated. This EFSM is used as input file for the test suite generator. The ultimate output file of the generator is also included in this appendix.

```lotos
specification connection_disconnection[LI,LO,UI,VO]:noexit

library NaturalNumber, DecNatRepr, Boolean endlib

type
L_Frame is
sorts L_Frame
opns SABM, VA, DISC, DM: -> L_Frame
endtype

type
V_Frame is
sorts V_Frame
opns ConReq, DiscReq: -> U_Frame
endtype

behaviour connection[LI,LO,UI,VO]

where

process connection[LI,LO,UI,VO]:noexit:=
  LI!DM; connect_res[LI,LO,UI,VO](Natnum(0))
  [ ]
  UI!Conreq; connect_res[LI,LO,UI,VO](Natnum(0))
  [ ]
  LI!DISC; LO!DM; connection[LI,LO,UI,VO]
  [ ]
  LI!SABM; (LO!DM; connection[LI,LO,UI,VO]
  [ ]
  LO!UA; disconnection[LI,LO,UI,VO])
endproc

process connect_res[LI,LO,UI,VO](tries:nat):noexit:=
  [tries ne NatNum(5)] -> (LO!SABM;
  (LI!SABM; LO!UA; disconnection[LI,LO,UI,VO]
  [ ]
```
LI!UA; disconnection[LI,LO,UI,UA]
[]
LI!DISC; LO!UA; connection[LI,LO,UI,UA]
[]
i; connect_res[LI,LO,UI,UA] (succ(tries)))
[]
[tries eq natnum(5)] -> LO!DM; connection[LI,LO,UI,UA]
end proc
process disconnection[LI,LO,UI,UA]:noexit=
UL!DiscReq; LO!DISC;
(LI!DM; connection[LI,LO,UI,UA]
[]
LI!UA; connection[LI,LO,UI,UA]
[]
LI!DISC; LO!UA; connection[LI,LO,UI,UA]
[]
LI!DISC; LO!UA; connection[LI,LO,UI,UA]
[]
i; LO!DM; connection[LI,LO,UI,UA]
end proc
endspec
This specification is generated by smile from the LOTOS specification specified on the previous pages. This EFSM is not used for the generation of the test suites, but the difference between this specification and the one generated by hand is only that this EFSM is using four ports and this EFSM contains one extra internal event. This EFSM is not optimized by the tool SMILE.

SPECIFICATION connection_disconnection [LI, LO, UI, UO]: noexit(* EFSM generated by SMILE *)

BEHAVIOUR
State1 [LI, LO, UI, UO]
WHERE

PROCESS State1 [LI, LO, UI, UO]: noexit :=
  LI !SABM;
  State2 [LI, LO, UI, UO]
[] LI !DISC;
  State9 [LI, LO, UI, UO]
[] UI !ConReq;
  State10 [LI, LO, UI, UO] (Natnum(0))
[] LI !DM;
  State10 [LI, LO, UI, UO] (Natnum(0))
ENDPROC

PROCESS State2 [LI, LO, UI, UO]: noexit :=
  LO !UA;
  State3 [LI, LO, UI, UO]
[] LO !DM;
  State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State3 [LI, LO, UI, UO]: noexit := (*!* 1 ;
  State4 [LI, LO, UI, UO]
[] LI !DISC;
  State5 [LI, LO, UI, UO]
[] UI !DiscReq;
  State6 [LI, LO, UI, UO]
ENDPROC

PROCESS State4 [LI, LO, UI, UO]: noexit :=
  LO !DM;
  State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State5 [LI, LO, UI, UO]: noexit :=
Using FDT LOTOS to derive tests

LO !UA;
   State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State6 [LI, LO, UI, UO]: noexit :=
   LO !DISC;
   State7 [LI, LO, UI, UO]
ENDPROC

PROCESS State7 [LI, LO, UI, UO]: noexit :=
   LI !DISC;
   State8 [LI, LO, UI, UO]
   [] LI !UA;
   State1 [LI, LO, UI, UO]
   [] LI !DM;
   State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State8 [LI, LO, UI, UO]: noexit :=
   LO !UA;
   State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State9 [LI, LO, UI, UO]: noexit :=
   LO !DM;
   State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State10 [LI, LO, UI, UO] (tries_71_0:nat): noexit :=
   [((tries_71_0 eq Natnum(5))] -> LO !DM;
   State1 [LI, LO, UI, UO]
   [] ([tries_71_0 ne Natnum(5))] -> LO !SABM;
   State11 [LI, LO, UI, UO] (tries_71_0)
ENDPROC

PROCESS State11 [LI, LO, UI, UO] (tries_71_1:nat): noexit :=
   (*!* i );
   State10 [LI, LO, UI, UO] (succ(tries_71_1))
   [] LI !DISC;
   State12 [LI, LO, UI, UO]
   [] LI !UA;
   State3 [LI, LO, UI, UO]
   [] LI !SABM;
   State13 [LI, LO, UI, UO]
ENDPROC

PROCESS State12 [LI, LO, UI, UO]: noexit :=
   LO !UA;
   State1 [LI, LO, UI, UO]
PROCESS State13 [LI, LO, UI, UO]: noexit :=
LO !UA;
   State3 [LI, LO, UI, UO]
ENDPROC

ENDSPEC
Specification of the EFSM derived by hand from the LOTOS specification:

SPECIFICATION connection[U,L] : noexit (*EFSM output *)

BEHAVIOUR S1 [U,L]

WHERE

PROCESS S1[U,L]: noexit :=
  L?DM; S2 [U,L](0) []
  U?ConReq; S2[U,L](0) []
  L?SABM; S3[U,L] []
  L?DISC; S4[U,L]
ENDPROC

PROCESS S2 [U,L] (x:nat): noexit :=
  [x lt N2] -> L!SABM; S5[U,L](x)
ENDPROC

PROCESS S3[U,L](x:nat): noexit :=
  L!DM; S1[U,L] []
  L!UA; S7[U,L]
ENDPROC

PROCESS S4[U,L]: noexit :=
  L!DM; S1[U,L]
ENDPROC

PROCESS S5[U,L] (x:nat): noexit :=
  L?DISC; S8[U,L] []
  L?UA; S7[U,L] []
  L?SABM; S6[U,L] []
  i; S2[U,L] (x+1)
ENDPROC

PROCESS S6[U,L] : noexit :=
  L!UA; S7[U,L]
ENDPROC

PROCESS S7[U,L]: noexit :=
  L?UA; S7[U,L] []
  L?DISC; S8[U,L] []
  L?SABM; S6[U,L] []
  U?DiscReq; S9[U,L] []
  L?DM; S1[U,L]
ENDPROC

PROCESS S8[U,L]: noexit :=
  L!UA; S1[U,L]
An example of a LOTOS specification, its EFSM and the derived Test Suite

```
ENDPROC

PROCESS S9[U,L]: noexit :=
  L!DISC; S10[U,L]
ENDPROC

PROCESS S10[U,L]: noexit :=
  L?DM; S1[U,L] []
  L?UA; S1[U,L]
ENDPROC
ENDSPEC
```
Figure 1. Extended Finite state machine in graphical form.
An example of a LOTOS specification, its EFSM and the derived Test Suite

EFSM read in a C-structure and with some added information for the derivation of the test suite.

SPECIFICATION OF: [CONNECTION, [U,L]]
WITH FIRST STATE: [S1, [U,L]]
number of states: 10

STATE : [1, stable, S1, [U,L]]
EVENT : [1, L?DM hu, inp] (S1) -> (S2)
EVENT : [2, U?CONREQ un, inp] (S1) -> (S2)
EVENT : [3, L?SABM hu, inp] (S1) -> (S3)
EVENT : [4, L?DISC hu, inp] (S1) -> (S4)

STATE : [2, unstable, S2, [U,L], X:NAT]
EVENT : [5, L!SABM un, outp] [XLTN2] (S2) -> (S5)

STATE : [3, unstable, S3, [U,L], X:NAT]
EVENT : [6, L!DM nu, outp] (S3) -> (S1)
EVENT : [7, L!UA nu, outp] (S3) -> (S7)

STATE : [4, unstable, S4, [U,L]]
EVENT : [8, L!DM nu, outp] (S4) -> (S1)

STATE : [5, stable, S5, [U,L], X:NAT]
EVENT : [10, L!UA nu, inp] (S5) -> (S7)
EVENT : [9, L?DISC nu, inp] (S5) -> (S8)
EVENT : [11, L?SABM nu, inp] (S5) -> (S6)
EVENT : [12, I un, int] (S5) -> (S2)

STATE : [6, unstable, S6, [U,L]]
EVENT : [13, L!UA nu, outp] (S6) -> (S7)

STATE : [7, stable, S7, [U,L]]
EVENT : [18, L!DM nu, inp] (S7) -> (S1)
EVENT : [14, L?UA nu, inp] (S7) -> (S7)
EVENT : [15, L?DISC nu, inp] (S7) -> (S8)
EVENT : [16, L?SABM nu, inp] (S7) -> (S6)
EVENT : [17, U?DISCREQ un, inp] (S7) -> (S9)

STATE : [8, unstable, S8, [U,L]]
EVENT : [19, L!UA hu, outp] (S8) -> (S1)

STATE : [9, unstable, S9, [U,L]]
EVENT : [20, L!DISC un, outp] (S9) -> (S10)

STATE : [10, stable, S10, [U,L]]
EVENT : [21, L?DM nu, inp] (S10) -> (S1)
EVENT : [22, L?UA hu, inp] (S10) -> (S1)
Generated Test Suite:

TTCN PATH NAME: (S1) - L?DM -> (S2)

L!DM
[ XLTN2 ] L?SABM
  I
[ XLTN2 ]
  L?SABM
  L!UA
  L!DM
  ?OTHERWISE
  ?EXPIRATION OF TIMER
  ?OTHERWISE
  ?EXPIRATION OF TIMER
  pass
  fail
  fail
  fail
  fail

TTCN PATH NAME: (S1) - U!CONREQ -> (S2)

U!CONREQ
[ XLTN2 ] L?SABM
  I
[ XLTN2 ]
  L?SABM
  L!UA
  L!DM
  ?OTHERWISE
  ?EXPIRATION OF TIMER
  ?OTHERWISE
  ?EXPIRATION OF TIMER
  pass
  fail
  fail
  fail
  fail

TTCN PATH NAME: (S1) - L?SABM -> (S3)

L!SABM
L?DM
[ XLTN2 ] U!CONREQ
  L?SABM
  L!UA
  L!DM
  ?OTHERWISE
  ?EXPIRATION OF TIMER
  pass
  fail
  fail
L?UA
U!DISCREQ
L?DISC
L!DM
?OTHERWISE
?EXPIRATION OF TIMER
pass
fail
fail
fail
fail
?EXPIRATION OF TIMER
fail
An example of a LOTOS specification, its EFSM and the derived Test Suite

TTCN PATH NAME: (S1) - L?DISC -> (S4)

L!DISC
L?DM
U!CONREQ
[ XLTN2 ]
L?SABM
L!UA
L!DM
pass
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail

TTCN PATH NAME: (S5) - L?UA -> (S7)

L!DM
[ XLTN2 ]
L?SABM
L!UA
START TIMER
?TIME OUT
U!DISCREQ
L?DISC
L!DM
pass
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
?OTHERWISE
fail
[ XLTN2 ]
L?SABM
L!UA
inconclusive
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail

TTCN PATH NAME: (S5) - L?DISC -> (S8)

L!DM
[ XLTN2 ]
L?SABM
L!DISC
L?UA
U!CONREQ
[ XLTN2 ]
L?SABM
L!UA
L!DM
pass
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
[ XLTN2 ]
L?SABM
L!UA
inconclusive
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
TTCN PATH NAME: (S5) - L?SABM -> (S6)

L!DM
[ XLTN2 ]
L?SABM
L!SABM
L?UA
U!DISCREQ
L?DISC
L!DM
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
?EXPIRATION OF TIMER

[ XLTN2 ]
L?SABM
L!UA
L!DM
?OTHERWISE
?EXPIRATION OF TIMER

TTCN PATH NAME: (S7) - L?DM -> (S1)

L!SABM
L?UA
L!DM
START TIMER
?TIME OUT
U!CONREQ
[ XLTN2 ]
L?SABM
L!UA
L!DM
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
L?DM
?OTHERWISE
?EXPIRATION OF TIMER

TTCN PATH NAME: (S7) - L?UA -> (S7)

L!SABM
L?UA
L!UA
START TIMER
?TIME OUT
U!DISCREQ
L?DISC
L!DM
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
L?DM
?OTHERWISE
?EXPIRATION OF TIMER

Using FDT LOTOS to derive tests
An example of a LOTOS specification, its EFSM and the derived Test Suite

TTCN PATH NAME: (S7) - L?DISC -> (S8)

L!SABM
L?UA
L!DISC
L?UA
U!CONREQ
L?SABM
L!UA
L!DM
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
?EXPIRATION OF TIMER
L?DM
?OTHERWISE
?EXPIRATION OF TIMER
pass
fail
fail
fail
fail
inconclusive
fail
fail
TTCN PATH NAME: (S7) - L?SABM -> (S6)

L!SABM
L?UA
L!SABM
L?UA
U!DISCREQ
L?DISC
L!DM
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
?EXPIRATION OF TIMER
L?DM
?OTHERWISE
?EXPIRATION OF TIMER
pass
fail
fail
fail
fail
inconclusive
fail
fail
TTCN PATH NAME: (S7) - U?DISCREQ -> (S9)

L!SABM
L?UA
U!DISCREQ
L?DISC
L!UA
U!CONREQ
L?SABM
L!UA
L!DM
?OTHERWISE
?EXPIRATION OF TIMER
?OTHERWISE
?EXPIRATION OF TIMER
L?DM
?OTHERWISE
?EXPIRATION OF TIMER
pass
fail
fail
fail
fail
inconclusive
fail
fail
TTCN PATH NAME: (S10) - L?DM -> (S1)

L!SABM
L?UA
U!DISCREQ
L?DISC
L!DM
START TIMER
?TIME OUT
U!CONREQ
L?SABM
L!UA
L!DM
pass
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
?OTHERWISE
fail
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
L?DM
?OTHERWISE
inconclusive
?EXPIRATION OF TIMER
fail

TTCN PATH NAME: (S10) - L?UA -> (S1)

L!SABM
L?UA
U!DISCREQ
L?DISC
L!UA
START TIMER
?TIME OUT
U!CONREQ
L?SABM
L!UA
L!DM
pass
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
?OTHERWISE
fail
?OTHERWISE
fail
?EXPIRATION OF TIMER
fail
L?DM
?OTHERWISE
inconclusive
?EXPIRATION OF TIMER
fail
?EXPIRATION OF TIMER
fail
Appendix 3 Specification of the LAPB protocol in LOTOS.

 specification lapb

 library boolean,naturalnumber,octet endlib

 type data is octet (* data is a queue of octets *)
sorts data
opns create : -> data
 add : octet,data -> data
 first : data -> octet
 error : -> octet
 eqns forall d,e:octet, z:buffer
 ofsort octet
 first(create) = error;
 first(add(d,z)) = d;

datatype buffer is data, naturalnumber (* define buffer for the sent data *)
sorts buffer
opns create : -> buffer (* empty buffer *)
 add : data,buffer -> buffer
 get_el : nat,buffer -> data (* for resending a frame *)
 first : buffer -> data
 remove : buffer -> buffer (* for acknowledgement *)
cut : nat,buffer -> buffer
 length : buffer -> nat (* to establish a buffer is full *)
eqns forall d,e: data y: nat, z: buffer
 ofsort data
 first(create) = create; (* empty octet queue *)
Using FDT LOTOS to derive tests

\[
\begin{align*}
\text{first}(\text{add}(d, \text{create})) &= d; \\
\text{first}(\text{add}(d, \text{add}(e, z))) &= \text{first}(\text{add}(e, z)); \\
\text{get}_e(l, z) &= \text{first}(z); \\
\text{get}_e(\text{succ}(y), z) &= \text{get}_e(y, \text{remove}(z)); \\
\text{ofsort} & \quad \text{buffer} \\
\text{remove}(&\text{create}) = \text{create}; \quad \text{(* empty buffer *)} \\
\text{remove}(\text{add}(d, \text{create}))) &= \text{create}; \\
\text{remove}(\text{add}(d, \text{add}(e, z))) &= \text{add}(d, \text{remove}(\text{add}(e, z))); \\
y \geq \text{length}(z) &\Rightarrow \text{cut}(y, z) = z; \quad \text{(* remove *)} \\
y \lt \text{length}(z) &\Rightarrow \text{cut}(y, z) = \text{cut}(y, \text{remove}(z)); \quad \text{(* till length le y *)} \\
\text{ofsort} & \quad \text{nat} \\
\text{length}(\text{create}) &= 0; \\
\text{length}(\text{add}(d, z)) &= \text{succ}(\text{length}(z)); \\
\end{align*}
\]

\text{endtype}

\text{type} \quad \text{mode is naturalnumber}

\text{sorts} \quad \text{mode}

\text{opns} \quad \text{basic, extended, none} \quad \rightarrow \text{mode}

\text{modulus} \quad \text{mode} \rightarrow \text{nat}

\text{k} \quad \rightarrow \text{nat}

\text{eqns} \quad \text{modulus}(\text{basic}) = \text{natnum}(8); \\
\text{modulus}(\text{extended}) = \text{natnum}(128); \\
\text{modulus}(\text{none}) = 0; \\
k = \text{natnum}(7); \quad \text{(* constant of LAPB protocol *)}

\text{endtype}

\text{type} \quad \text{frame_types}

\text{sorts} \quad \text{sort, name}

\text{opns} \quad \text{com, res} \quad \rightarrow \text{sort}

\text{I, RR, RNR, REJ} \quad \rightarrow \text{name}

\text{endtype}

\text{behaviour} \quad \text{information_transfer: noexit}

\text{process} \quad \text{information_transfer} \quad \text{[prt_in, prt_out, info_in, info_out, busy, error]}: \quad \text{noexit}:=

\text{hide} \quad \text{poll, reject, confirm, resend, ack, start, stop, timeout in}

\text{receive_frame} \quad \text{[prt_in, info_out, ack, confirm, reject, poll, resend, wait]} \quad (0)

\mid \text{poll, reject, confirm, resend, ack, wait} \mid

\mid (\text{send_buffer} \quad \text{[info_in, error, ack, resend, frame, start, stop]} \quad (\text{create}, 0, 0) \mid

\mid \text{frame} \mid

\text{send_frame} \quad \text{[prt_out, busy, poll, reject, confirm, wait, start, timeout]} \quad (0, f) \mid

\mid \text{[start, stop, timeout]} \mid

\text{timer} \quad \text{[error, start, stop, timeout]} \quad (0))

\text{where}

\text{process} \quad \text{timer} \quad \text{[error, start, stop, timeout]} \quad \text{(tries: nat)}: \quad \text{noexit}:=

\mid \text{tries} < \text{N2} \rightarrow \text{start}; \\
\mid \text{(stop; timer}(0)) \\
\mid \\
\mid \text{i; timeout; timer}(\text{tries} + 1) \\
\mid \\
\mid
timer(tries))
[]
  [tries = \text{N}2] \rightarrow \text{error; stop}
endproc timer

process send_buffer[info\text{in}, error, ack, resend, frame, start, stop] (buf:buffer, Vs, el:fnum)
  [el \leq \text{length}(buf)] \rightarrow \text{snd!get}_\text{el}(el,buf)!Vs; send_buffer(buf,\text{succ}(Vs), \text{succ}(el))
  []
  [\text{length}(buf) < k] \rightarrow \text{Info\text{in}?d:}\text{data}; \text{send_buffer}\text{ add}(d, buf), Vs, el)
  []
  \text{ackn?ns:nat};
    [\text{between}(ns, Vs, Vs + \text{length}(buf))] \rightarrow \text{stop;}
    ([ns \neq (Vs + \text{length}(buf)) \mod m] \rightarrow \text{start;})
    \text{send_buffer}(\text{cut}((Vs - ns, buf), Vs, (Vs - ns))
      []
    [ns \equiv Vs] \rightarrow \text{send_buffer}(buf, Vs, el)
      []
  [\text{between}(ns, Vs + \text{length}(buf), Vs - 1)] \rightarrow \text{error; stop}
  []
  \text{resend?ns:nat;}
    ([\text{between}(ns, Vs - 1, Vs + \text{length}(buf))] \rightarrow \text{stop;}
    \text{send_buffer}(	ext{cut}(Vs - ns, buf), ns, 0)
      []
    [\text{between}(ns, (Vs + \text{length}(buf)), Vs - 1)] \rightarrow \text{error; stop}
endproc

process send_frame [prt\text{out}, busy, poll, reject, confirm, wait, start, timeout]
  (Vr:fnum, b: name): \text{noexit}:=
    \text{confirm?Vr; send_frame}(Vr, b)
  []
    i; \text{prt\text{out}!com!b!Vr!false; send_frame}(Vr, b)
      []
    \text{frame?Vs:fnum?d:}\text{data}; \text{prt\text{out}!com!!Vr!false!Vs!d}; \text{start}; \text{send_frame}(Vr, b)
      [> \text{send_s_frame}(false, b); \text{accept } b \text{ in } \text{send_frame}(Vr, b)]

where

process send_s_frame[busy, poll, reject, wait, timeout](w:bool, b: name): \text{exit}:=
  (\text{timeout}; \text{prt\text{out}!com!b!Vr!true; start}
    []
  \text{poll}; \text{prt\text{out}!res!b!Vr!true}
    []
  \text{busy?b:}\text{name}; \text{prt\text{out}!com!b!Vr!false}
    []
  [w=true] \rightarrow \text{confirm?Vr; \text{prt\text{out}!com!b!Vr!false}
    []
  \text{wait?w:bool};
  [w=false] \rightarrow \text{exit}(b) [] [w=true] \rightarrow \text{send_s_frame}(w, b)
    []
  \text{reject; \text{prt\text{out}!com!REJ!Vr!false; start;}
    [w=false] \rightarrow \text{exit}(RR) [] [w=true] \rightarrow \text{send_s_frame}(w, RR);
endproc send_s_frame
endproc send_frame

process rec_frame[prt_in,info_out,ack,confirm,reject,poll,resend,wait](Vr:nat):noexit :=
    ([p] -> poll);
    [ns eq Vr] -> Info_out!d; confirm!Vr; rec_frame (succ(Vr))
        []
    [ns ne Vr] -> reject!Vr; rec_frame (Vr)
        []
        []
        []
    prtl!in?s:sort?RNR?nr:nat?p:bool; ackn!nr; wait!true);
    ([s = com and p] -> poll); rec_frame(Vr)
endproc rec_frame
endspec
Appendix 4  An other example of a disconnection connection Specification, with the generated EFSM.

specification connection_disconnection[LI,LO,UI,UO]:noexit

library NaturalNumber, DecNatRepr, Boolean endlib

type verdict is Boolean
sorts   verdict
opns   pass,fail : -> verdict
       _eq_ : verdict, verdict -> Bool
eqns   ofsort Bool
       pass eq pass = true;
       fail eq fail = true;
endtype

type L_Frame is
sorts   L_Frame
opns   SABM,UA,DISC,DM : -> L_Frame
endtype

type U_Frame is
sorts   U_Frame
opns   ConReq, DiscReq: -> U_Frame
endtype

behaviour disc_con[LI,LO,UI,UO]

where

process disc_con[LI,LO,UI,UO]:noexit:=
  connection[LI,LO,UI,UO] >> accept result:verdict in
  [result eq pass] -> (disconnection[LI,LO,UI,UO] >> disc_con[LI,LO,UI,UO])
  []
  [result eq fail] -> disc_con[LI,LO,UI,UO]

where

process connect_res[LI,LO,UI,UO](tries:nat):exit(verdict):=
  [tries ne NatNum(5)] -> (LO!SABM;
   (LI!SABM; LO!UA; exit(pass)
    []
   LI!UA; exit(pass)
    []

using FDT LOTOS to derive tests
An other example of a specification, with the generated EFSM.

**SPECIFICATION connection_disconnection [LI, LO, UI, UO]: noexit(‘* EFSM generated by SMILE ‘)**

**BEHAVIOUR**

State1 [LI, LO, UI, UO]

WHERE

```
PROCESS State1 [LI, LO, UI, UO]: noexit :=
LI !IDM;
   State2 [LI, LO, UI, UO] (NatNum(0))
[LI !ConReq;
   State2 [LI, LO, UI, UO] (NatNum(0))
[LI !DISC;
   State21 [LI, LO, UI, UO]
[LI !SABM;
   State7 [LI, LO, UI, UO]
ENDPROC

PROCESS State2 [LI, LO, UI, UO] (tries_72_0:nat): noexit :=
[(tries_72_0 ne NatNum(5)) -> LO !SABM;
   State3 [LI, LO, UI, UO] (tries_72_0)
[![tries_72_0 eq NatNum(5)] -> LO !IDM;
   State36 [LI, LO, UI, UO]
ENDPROC

PROCESS State3 [LI, LO, UI, UO] (tries_72_1:nat): noexit :=
LI !SABM;
   State4 [LI, LO, UI, UO]
[LI !UA;
   State33 [LI, LO, UI, UO]
[LI !DISC;
   State34 [LI, LO, UI, UO]
[!!] i ;
   State2 [LI, LO, UI, UO] (succ(tries_72_1))
ENDPROC

PROCESS State4 [LI, LO, UI, UO]: noexit :=
LO !UA;
   State5 [LI, LO, UI, UO]
ENDPROC

PROCESS State5 [LI, LO, UI, UO]: noexit :=
(‘*exit !pass‘) i ;
   State6 [LI, LO, UI, UO] (pass)
ENDPROC

PROCESS State6 [LI, LO, UI, UO] (result_73_0:verdict): noexit :=
[(result_73_0 eq fail) -> LI !SABM;
   State7 [LI, LO, UI, UO]
```
[] [(result_73_0 eq fail)] -> LI !DISC;
  State21 [LI, LO, UI, UO]
[] [(result_73_0 eq fail)] -> UI !ConReq;
  State2 [LI, LO, UI, UO] (NatNum(0))
[] [(result_73_0 eq fail)] -> LI !DM;
  State2 [LI, LO, UI, UO] (NatNum(0))
[ ( "i" ) [(result_73_0 eq pass)] -> i ;
  State23 [LI, LO, UI, UO]
[] [(result_73_0 eq pass)] -> LI !DISC;
  State25 [LI, LO, UI, UO]
[] [(result_73_0 eq pass)] -> UI !DiscReq;
  State27 [LI, LO, UI, UO]
ENDPROC

PROCESS State7 [LI, LO, UI, UO]: noexit :=
  LO !DM;
  State8 [LI, LO, UI, UO]
[] LO !UA;
  State9 [LI, LO, UI, UO]
ENDPROC

PROCESS State8 [LI, LO, UI, UO]: noexit :=
  (*exit !fail*) i ;
  State6 [LI, LO, UI, UO] (fail)
ENDPROC

PROCESS State9 [LI, LO, UI, UO]: noexit :=
  (*i*) i ;
  State10 [LI, LO, UI, UO]
[] LI !DISC;
  State13 [LI, LO, UI, UO]
[] UI !DiscReq;
  State15 [LI, LO, UI, UO]
ENDPROC

PROCESS State10 [LI, LO, UI, UO]: noexit :=
  LO !DM;
  State11 [LI, LO, UI, UO]
ENDPROC

PROCESS State11 [LI, LO, UI, UO]: noexit :=
  (*exit*) i ;
  State12 [LI, LO, UI, UO]
ENDPROC

PROCESS State12 [LI, LO, UI, UO]: noexit :=
  (*exit !fail*) i ;
  State6 [LI, LO, UI, UO] (fail)
ENDPROC
An other example of a specification, with the generated EFSM.

PROCESS State13 [LI, LO, UI, UO]: noexit :=
    LO !UA;
    State14 [LI, LO, UI, UO]
ENDPROC

PROCESS State14 [LI, LO, UI, UO]: noexit :=
    (*"exit") i ;
    State12 [LI, LO, UI, UO]
ENDPROC

PROCESS State15 [LI, LO, UI, UO]: noexit :=
    LO !DISC;
    State16 [LI, LO, UI, UO]
ENDPROC

PROCESS State16 [LI, LO, UI, UO]: noexit :=
    LI !DISC;
    State17 [LI, LO, UI, UO]
[] LI !UA;
    State19 [LI, LO, UI, UO]
[] LI !DM;
    State20 [LI, LO, UI, UO]
ENDPROC

PROCESS State17 [LI, LO, UI, UO]: noexit :=
    LO !UA;
    State18 [LI, LO, UI, UO]
ENDPROC

PROCESS State18 [LI, LO, UI, UO]: noexit :=
    (*"exit") i ;
    State12 [LI, LO, UI, UO]
ENDPROC

PROCESS State19 [LI, LO, UI, UO]: noexit :=
    (*"exit") i ;
    State12 [LI, LO, UI, UO]
ENDPROC

PROCESS State20 [LI, LO, UI, UO]: noexit :=
    (*"exit") i ;
    State12 [LI, LO, UI, UO]
ENDPROC

PROCESS State21 [LI, LO, UI, UO]: noexit :=
    LO !DM;
    State22 [LI, LO, UI, UO]
ENDPROC
Using FDT LOTOS to derive tests

PROCESS State22 [LI, LO, UI, UO]: noexit := (*exit !fail*) i ;
    State6 [LI, LO, UI, UO] (fail)
ENDPROC

PROCESS State23 [LI, LO, UI, UO]: noexit :=
    LO !DM;
    State24 [LI, LO, UI, UO]
ENDPROC

PROCESS State24 [LI, LO, UI, UO]: noexit :=
    (*exit*) i ;
    State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State25 [LI, LO, UI, UO]: noexit :=
    LO !UA;
    State26 [LI, LO, UI, UO]
ENDPROC

PROCESS State26 [LI, LO, UI, UO]: noexit :=
    (*exit*) i ;
    State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State27 [LI, LO, UI, UO]: noexit :=
    LO !DISC;
    State28 [LI, LO, UI, UO]
ENDPROC

PROCESS State28 [LI, LO, UI, UO]: noexit :=
    LI !DM;
    State29 [LI, LO, UI, UO]
[] LI !UA;
    State30 [LI, LO, UI, UO]
[] LI !DISC;
    State31 [LI, LO, UI, UO]
ENDPROC

PROCESS State29 [LI, LO, UI, UO]: noexit :=
    (*exit*) i ;
    State1 [LI, LO, UI, UO]
ENDPROC

PROCESS State30 [LI, LO, UI, UO]: noexit :=
    (*exit*) i ;
    State1 [LI, LO, UI, UO]
ENDPROC
PROCESS State31 [LI, LO, UI, UO]: noexit := LO !UA;
    State32 [LI, LO, UI, UO]
ENDPROC

PROCESS State32 [LI, LO, UI, UO]: noexit := (*exit*) i ;
    State31 [LI, LO, UI, UO]
ENDPROC

PROCESS State33 [LI, LO, UI, UO]: noexit := (*exit !pass*) i ;
    State6 [LI, LO, UI, UO] (pass)
ENDPROC

PROCESS State34 [LI, LO, UI, UO]: noexit := LO !UA;
    State35 [LI, LO, UI, UO]
ENDPROC

PROCESS State35 [LI, LO, UI, UO]: noexit := (*exit !fail*) i ;
    State6 [LI, LO, UI, UO] (fail)
ENDPROC

PROCESS State36 [LI, LO, UI, UO]: noexit := (*exit !fail*) i ;
    State6 [LI, LO, UI, UO] (fail)
ENDPROC

ENDSPEC
Appendix 5 Source Text of the program Test Suite Generator

/***************************************************************************/
/* FILE TYPEDEF.H */
/***************************************************************************/
/* type definition for the pointer structure of the extended finite state */
/* machine and the types for the derived ttcn traces from the efsm */
/***************************************************************************/
typedef enum {yes, half, no} yes_no; /* to indicate uniqueness */
typedef enum {pass, fail, inconclusive, none} verdict_type;
typedef enum {input, output, internal, syn} sort_type; /* action sort */
typedef enum {stable, instable} stable_type; /* stability of a state */

/***************************************************************************/
/* definition of state type containing: */
/* name: the name of the state */
/* port: list of ports via which this state can communicate */
/* var: list of variables which are given with this state */
/***************************************************************************/

struct state_type
{
    char name[80], port[80], var[80];
};

/***************************************************************************/
/* definition of event type, containing the next variables */
/* to: state_type used for variable transfer, relabelling */
/* event: name of event, rename into action */
/* cond: the precondition of the action */
/* sort: input, output, internal or synchronization */
/* unique: is this event unique, half unique or not unique */
/* num: event number */
/* next: next event which has the same from state */
/* next_from: next event with the same to state */
/* fp: pointer to the from state */
/* tp: pointer to the to state */
/***************************************************************************/

struct event_type
{
    struct state_type *to;
    char event[80], cond[80];
    sort_type sort;
    yes_no unique;
    int num;
    struct event_type *next, *next_from;
    struct ext_state_type *fp, *tp;
};
Using FDT LOTOS to derive tests

/* definition of extended state type containing:
   * state: to the state type containing name, ports and variables
   * num: state number
   * sort: if the event is stable or not stable
   * next: next state of machine m with num + 1
   * first: first event which begins in this state
   * first_from: first event which end in this state
*/

struct ext_state_type
{
    struct state_type *state;
    int num;
    stable_type sort;
    struct ext_state_type *next;
    struct event_type *first, *first_from;
};

/* extended finite state machine type containing the machine, it’s name
   * number of state and its first state
*/

struct efsm_type
{
    struct state_type *name, *first_state;
    struct ext_state_type *first;
    int num_states;
};

/* TYPES USED FOR TTCGEN
*/

/* definition of ttcn element type. every element contains:
   * cont: contents in text, for example ?OTHERWISE
   * label: giving an element a label to jump in a path
   * verdict: pass, fail, inconclusive or none
   * ev: pointing to the event which has to be executed
   * nl: next level, the events which have to be executed after this event
   * np: next possibility, next alternative event
*/

struct ttcn_el_type
{
    char cont[40], label[5], comment[20];
    verdict_type verdict;
    struct event_type *ev;
    struct ttcn_el_type *nl,*np;
};

/* definition of ttcn path type. every path contains:
   * name: name of the path, normally the state and event to be tested
   * fel: first element of the path
   * next: next path
*/

struct ttcn_path_type
{
    char name[80];

struct ttcn_el_type *fel; /*first element */
struct ttcn_path_type *next;

/* definition of state and event list. This is only temporary used to */
/* indicate if every state or event is used. */

struct state_list_type
{                          
    struct ext_state_type  *state;
    struct state_list_type *next;
};

struct event_list_type
{                         
    struct event_type     *ev;
    struct event_list_type *next;
};

/* END OF TYPE DEFINITION */

/* END OF TYPE DEFINITION */
Using FDT LOTOS to derive tests

/* MODULE EFSMPROC.C */

/***************************************************************************/
/* This module will read the input file in LOTOS format and will result a */
/* pointer structure as output. The main function is called read_smile(in)*/
/***************************************************************************/
/* The module also consists of the following 20 functions */
/* str_form: to format strings */
/* action_sort: to detect the type of an action */
/* search_state: searching name in state_list */
/* scan_state, scan_event, read_state, read_efsm: */
/* to read the read the input file */
/* add_inverse_path: to add the inverse path to the efsm */
/* add_list, count_states, full_list, swap_from_events, find_init_step, */
/* swap_to_events, find_term_step: for finding the shortest initiation */
/* and termination step */
/* add_non_unique_event, add_half_unique_event, search_unique_events: */
/* to define the uniqueness of an event */
/* empty_state_list, empty_event_list: to free memory space */
/***************************************************************************/

/* include <stdio.h> */
/* include "typedef.h" */

/***************************************************************************/
/* function to give string s the standard format and to delete unknown */
/* and unwanted signs */
/***************************************************************************/
str_form(s)
char *s;
{
    int p =0;
    int q =0;
    char *val = "013456789();<>?!-,";
    while ( s[p] != '\0' )
    {
        s[q] = toupper(s[p]);
        if (isalpha(s[p]) || (chr(val, s[q]) != NULL)) q++;
        p++;
    }
    s[q] = '\0';
} /* str_form */

/***************************************************************************/
/* function to detect the event’s sort, it will return with the sort */
/***************************************************************************/
sort_type action_sort(s)
char *s;
{
    if (strchr(s,'?') != NULL) return(input);
    else
        { 
            if (strchr(s,'!') != NULL) return(output);
            else
                { 
                    if (strcmp(s,"I") == 0) return(internal);
                    else return(syn);
                }
        }
}
struct ext_state_type *search_state(m, s)
struct efsm_type *m;
struct state_type *s;
{
    struct ext_state_type *st;
st = m->first;
    while (strcmp(st->state->name, s->name) != 0)
    {
        st = st->next;
        if (st == NULL)
            {
                printf("State name not found: %s\n", s->name);
                return(NULL);
            }
    }
    return(st);
} /* search_state */

struct state_type *scan_state(in)
FILE *in;
{
    struct state_type *st;
    char ch;
    char *val=";\n\n[\n{\nwhile (((ch = fgetc(in)) == ' ') || (ch == '\n')));
    st = (struct state_type *)malloc(sizeof(struct state_type));
    strcpy(st->name,"");
    while ((ch != '[') && (ch != EOF)) \n        {
            strcat(st->name,&ch);
            ch = fgetc(in);
        }
    while (((ch != ']') && (ch != EOF)) ch = fgetc(in);
    while (((ch = fgetc(in)) == EOF))
        {
            strcat(st->port,&ch);
        }
    while (strchr(val,(ch = fgetc(in))) == NULL);
    if (ch == '
')
        {
            while (((ch = fgetc(in)) == EOF)) strcat(st->var,&ch);
            while (strchr(val,(ch = fgetc(in))) == NULL);
        } ungetc(ch,in);
    if (feof(in))
printf("out of data while reading state\n ");
return(NULL);
}
str_form(st->name);
str_form(st->var);
str_form(st->port);
return(st);
} /* scan_state */

 /**<**************************************************************************/
  /**< function to read all aspects of an event from the file *in*
  /**< it returns with the pointer to the read event or with the pointer NULL */
  /**< in case of a syntax error */
  /**<**************************************************************************/

struct event_type *scan_event (in)
FILE *in;
{
struct event_type *ev;
char ch;
ev = (struct event_type *)malloc(sizeof(struct event_type));
while (((ch = fgetc(in)) == ' ') || (ch == '\n')));
if (ch != '\n') {
  while (((ch = fgetc(in)) != ']') && (ch != EOF)) strcat(ev->cond,&ch);
  while (((ch = fgetc(in)) != '>') && (ch != EOF));
  while (((ch = fgetc(in)) == ' ') || (ch == '\n')));
while ((ch != ';') && (ch != EOF)) {
  strcat(ev->event,&ch);
  ch = fgetc(in);
}
ev->to = scan_state(in);
if (ch == EOF) {
  printf("out of data while reading event \n");
  return(NULL);
} str_form(ev->event);
ev->sort = action_sort(ev->event);
str_form(ev->cond);
return(ev);
}

 /**<**************************************************************************/
  /**< function to read the whole description within a state defined as a */
  /**< process in efsm_lotos (states and it’s related events) from the file in*/
  /**< returns with a pointer to the read state or NULL in case of an error */
  /**<**************************************************************************/

struct ext_state_type *read_state(in,n,en)
FILE *in;
int n, *en;
{
struct ext_state_type *st;
struct event_type *ev;
char ch,s[80];
do
[ fscanf(in,"%s",s);

while ((strstr(s,"PROCESS") == NULL) && (strstr(s,"ENDSPEC") == NULL) && (s[0] != '\0'));
if (strstr(s,"ENDSPEC") != NULL) return(NULL);
if (s[0] == '\0')
{
    printf("keyword PROCESS or ENDSPEC not found");
    return(NULL);
}
else
{
    st = (struct ext_state_type *)malloc(sizeof(struct ext_state_type));
st->num = n;
st->sort = stable;
st->state = scan_state(in);
if (st->state == NULL) return(NULL);
else
{
    while (fgetc(in) != '\n');
st->first = scan_event (in);
ev = st->first;
if (ev->sort == output) st->sort = instable;
ev->num = (*en)++;
ev->fp = st;
ev->unique = yes; /* initialize event as unique */
fscanf(in,"%s",s);
while (strcmp(s,\[J") 0)
{
    if ((ev->next = scan_event (in)) != NULL)
    {
        ev = ev->next;
        if (ev->sort == output) st->sort = instable;
ev->fp = st;
ev->num = (*en)++;
ev->unique = yes;
        else return(NULL);
fscanf(in,"%s",s);
    }
if (strcmp(s,"ENDPROC") == 0)
    {
        return(st);
    }
else
{
    printf("keyword ENDPROC not found");
    return(NULL);
}
}
} /* read_state */

/**************************************************************************/
struct efsm_type *read_efsm(in)
FILE *in;

/*****************************/
/* function to read whole the efsm-lotos specification from the file in */
/* returns with a pointer to the read machine or NULL in case of an error */
/*****************************/
{  
  struct efsm_type *efsm;
  struct ext_state_type *p;
  struct event_type *ev;
  char ch,s[80];
  int en = 1, n = 2;
  do
    fscanf(in,"%s",s);
  while (strcmp(s,"SPECIFICATION") != 0);
  efsm = (struct efsm_type *)malloc(sizeof(struct efsm_type));
  efsm->name = scan_state(in);
  do
    fscanf(in,"%s",s);
  while (strcmp(s,"BEHAVIOUR") != 0);
  efsm->first_state = scan_state(in);
  efsm->first = read_state(in,1,&en);
  p = efsm->first;
  while ((p->next = read_state(in,n++,&en)) != NULL)
    {
      p = p->next;
    }
  p = efsm->first;
  while (p != NULL)
    {
      ev = p->first;
      while (ev != NULL)
        {
          nst = ev->tp;
          if (nst->first_from == NULL) nst->first_from = ev;
          else
            {
              ev_from = nst->first_from;
              while (ev_from->next_from != NULL) ev_from = ev_from->next_from;
              ev_from->next_from = ev;
            }
          p = p->next;
        }
      return (efsm);
    }
} /* read_efsm */

/*****************************************************************************/
/* function to add the inverse path to a read efsm which contains only */
/* the forward path */
/*****************************************************************************/
add_inverse_path(e)
struct efsm_type *e;
{
  struct ext_state_type *st, *nst;
  struct event_type *ev, *ev_from;
  st = e->first;
  while (st != NULL)
    {
      ev = st->first;
      while (ev != NULL)
        {
          nst = ev->tp;
          if (nst->first_from == NULL) nst->first_from = ev;
          else
            {
              ev_from = nst->first_from;
              while (ev_from->next_from != NULL) ev_from = ev_from->next_from;
              ev_from->next_from = ev;
            }
        }
      p = p->next;
    }
  return(e);
}
int add_list(list, st)
struct state_list_type *list;
struct ext_state_type *st;
{
    struct state_list_type *n, *s;
    n = list;
    while ((n->state != st) && (n->next != NULL))
    {
        n = n->next;
    }
    if (n->state != st)
    {
        s = (struct state_list_type *)malloc(sizeof(struct state_list_type));
        s->state = st;
        n->next = s;
        return(1);
    }
    else return(0);
} /* add_list */

int count_states(e)
struct efsm_type *e;
{
    int num = 1;
    struct ext_state_type *st;
    st = e->first;
    while (((st = st->next) != NULL) num++;
    e->num_states = num;
} /* count_states */

full_list(list, e)
struct state_list_type *list;
struct efsm_type *e;
{
    int num = 1;
    struct state_list_type *st;
    st = list;
    while (((st = st->next) != NULL) num++;
}
Using FDT LOTOS to derive tests

```c
return (num >= e->num_states);
) /* full_list */

/** function to place event ev as the first event of state st, this first */
/** element will indicate the shortest path to the initial state */
/****************************************************************************/

swap_from_events(st,ev)
struct ext_state_type *st;
struct event_type *ev;
{
    struct event_type *evs;
    evs = st->first_from;
    if (evs != ev)
    {
        while (evs->next_from != ev) evs = evs->next_from;
        evs->next_from = evs->next_from;
        ev->next_from = st->first_from;
        st->first_from = ev;
    }
} /* swap events */

/** this function will change the efsm in such a way that all the */
/** first_from events will point at the shorts path to the initial state */
/****************************************************************************/

find_init_step(e)
struct efsm_type *e;
{
    struct state_list_type *list,*lp;
    struct ext_state_type *st;
    struct event_type *ev;
    st = e->first;
    list = ( struct state_list_type *)malloc(sizeof(struct state_list_type));
    list->next = NULL;
    list->state = st;
    lp = list;
    while (!full_list(list,e))
    {
        ev = st->first;
        while (ev != NULL)
        {
            if (add_list(list,ev->tp)) swap_from_events(ev->tp,ev);
            ev = ev->next;
        }
        lp = lp->next;
        st = lp->state;
    }
} /* find_init_step */

/** the same as for the init path, but now for the termination path */
/****************************************************************************/

swap_to_events(st,ev)
struct ext_state_type *st;
struct event_type *ev;
{

```
struct event_type *evs;
evs = st->first;
if (evs != ev)
{
    while (evs->next != ev) evs = evs->next;
    evs->next = ev->next;
    ev->next = st->first;
    st->first = ev;
}
} /* swap events */

/***************************************************************************/
/* the first events will point at the shortest path to the final state */
/***************************************************************************/

find_term_step(e)
struct efsm_type *e;
{
    struct state_list_type *list,*lp;
    struct ext_state_type *st;
    struct event_type *ev;
    st = e->first;
    list = ( struct state_list_type *) malloc(sizeof(struct state_list_type));
    list->next = NULL;
    list->state = st;
    lp = list;
    while (!full_list(list,e))
    {
        ev = st->first_from;
        while (ev != NULL)
        {
            if (add_list(list,ev->fp,ev) || swap_to_events(ev->fp,ev);
            ev = ev->next_from;
        }
        lp = lp->next;
        st = lp->state;
    }
} /* find_term_step */

/***************************************************************************/
/* searching non and half unique actions for the function search_unique */
/* events. It will add every new event to a event_list and will change */
/* the unique variable if a double element is found */
/***************************************************************************/

struct event_list_type *add_non_unique_event(l,e)
struct event_list_type *l;
struct event_type *e;
{
    struct event_list_type *lp,*olp;
    lp = l; olp = NULL;
    while (lp != NULL)
    {
        if (strcmp(lp->ev->event,e->event) == 0)
        {
            if (lp->ev->tp == e->tp)
            {
                lp->ev->unique = no;
                e->unique = no;
                return (1);
            }
        }
    }
}
Using FDT LOTOS to derive tests

```c
struct event_list_type *add_half_unique_event(l, e)
struct event_list_type *l;
struct event_type *e;
{
    struct event_list_type *lp, *olp;
    lp = l; olp = NULL;
    while (lp != NULL)
    {
        if (strcmp(lp->ev->event, e->event) == 0)
        {
            lp->ev->unique = half;
            e->unique = half;
            return (l);
        }
        olp = lp;
        lp = lp->next;
    }
    lp = (struct event_list_type *)malloc(sizeof(struct event_list_type));
    lp->ev = e;
    lp->next = NULL;
    if (olp != NULL) olp->next = lp;
    else l = lp;
    return(l);
}
```

```c
struct event_list_type *empty_event_list(l)
struct event_list_type *l;
{
    struct event_list_type *o;
    while (l != NULL)
    {
        o = l;
        l = l->next;
        free(o);
    }
    return(NULL);
}
```

```c
empty_state_list(l)
struct state_list_type *l;
{
    struct state_list_type *o;
    while (l != NULL)
    {
```
lo = l;
  l = l->next;
  free(lo);
}

/***************************************************************************/
/* function to add the information if an event is unique, half unique or */
/* not unique. these states have the following meaning: */
/* unique: there is no event with the same name */
/* half unique: there are no events with the same name and destination */
/* non unique: there are events with the same name and destination */
/* this function is used to find a unique verification path */
/***************************************************************************/

search_unique_events(m)
struct efsm_type *m;
{
  struct event_list_type *el,*eln;
  struct event_type *ev;
  struct ext_state_type *st;
  st = m->first;
  el = NULL;
  while (st != NULL)
  {
    ev = st->first;
    while (ev != NULL)
    {
      el = add_half_unique_event(el,ev);
      ev = ev->next;
    }
    st = st->next;
  }
  el = empty_event_list(el);
  st = m->first;
  while (st != NULL)
  {
    ev = st->first;
    while (ev != NULL)
    {
      el = add_non_unique_event(el,ev);
      ev = ev->next;
    }
    st = st->next;
  }
  empty_event_list(el);
}

/***************************************************************************/
/* read_smile will read the efsm and add all the specific information */
/* which is needed to derive test cases. It returns a pointer to the */
/* machine */
/***************************************************************************/

struct efsm_type *read_smile(in)
FILE in;
{
  struct efsm_type *m;
  m = read_efsm(in);
  count_states(m);
  add_inverse_path(m);
find_init_step(m);
find_term_step(m);
search_unique_events(m);
return(m);

/**
 * END OF MODULE EFSMPROC.C
 */
/**
 * END OF MODULE EFSMPROC.C
 */
/*
 * MODULE TTCNGEN.C
 *
 * ttngen module will generate for every input event a test trace, containing
 * the four steps initialization, evaluation, verification and termination.
 * The function gen_test_suite() will return this test suite.
 */

/*
 * This module contains the next 9 functions:
 * create_empty_element, create_init_path, create_term_path,
 * create_side_path, create_verification_path, create_reception_path,
 * create_eval_path, create_path, gen_test_suite
 */

#include <stdio.h>
#include "typedef.h"

/* declaration of functions because of recursion */

struct ttcn_el_type *create_reception_path();
struct ttcn_el_type *create_side_path();

/* initialization of an empty ttcn element */

struct ttcn_el_type *create_empty_element();
{
    struct ttcn_el_type *elem;
    elem = (struct ttcn_el_type *)malloc(sizeof(struct ttcn_el_type));
    elem->ev = NULL;
    elem->nl = NULL;
    elem->np = NULL;
    elem->verdict = none;
    strcpy(elem->cont, "") ;
    strcpy(elem->label, "") ;
    return (elem);
}

/* create ttcn initialization path from state st to state est. It will */
/* return with a pointer to this path */

struct ttcn_el_type *create_init_path(m, st)
struct efsm_type *m; /* machine */
struct ext_state_type *st; /* actual state */
{
    struct ttcn_el_type *elem, *nelem; /* element */
    if (m->first == st) return (NULL); /* machine already in st */
    if (st->first_from->fp != m->first) /* search shortest init path */
    {
        elem = create_init_path(m, st->first_from->fp);
        nelem = elem;
        while (nelem->nl != NULL) nelem = nelem->nl;
        nelem->nl = create_empty_element();
        nelem = nelem->nl;
        nelem->ev = st->first_from;
        return (elem);
    } else
    {
        elem = create_init_path(m, st->first_from->fp);
        nelem = elem;
        while (nelem->nl != NULL) nelem = nelem->nl;
        nelem->nl = create_empty_element();
        nelem = nelem->nl;
        nelem->ev = st->first_from;
        return (elem);
    }
}
Using FDT LOTOS to derive tests

```c
/* create ttcn termination path from state st to state m->first. It will */
/* return with a pointer to the first element of this path, with final */
/* verdict verdict */
/**************************************************************************/
struct ttcn_el_type *create_term_path(m, st, verdict)
struct efsm_type *m;
struct ext_state_type *st;
verdict_type verdict;
{
    struct ttcn_el_type *elem; /* previous element */
    if (st != m->first) /* search shortest term path */
    {
        elem = create_empty_element();
        elem->ev = st->first;
        elem->nl = create_term_path(m, st->first->tp, verdict);
        if (elem->nl == NULL)
            elem->verdict = verdict;
        return(elem);
    }
    else
        return(NULL);
}
/***************************************************************************/
/* create the paths which are existing because of internal events and */
/* non determinism. also the possibilities of otherwise and expiration of */
/* the timer will be added (only if pass_event is an output) */
***************************************************************************/
struct ttcn_el_type *create_side_path(m, ev, pass_ev, verdict)
struct efsm_type *m;
struct event_type *ev, *pass_ev;
verdict_type verdict;
{
    struct ttcn_el_type *elem, *el2;
    elem = NULL;
    if (ev != NULL)
    {
        if ((ev->sort == output) && (ev != pass_ev))
        {
            /* if event is an other possibility */
            elem = create_empty_element();
            elem->ev = ev;
            elem->nl = create_term_path(m, ev->tp, verdict);
            if (elem->nl == NULL) elem->verdict = inconclusive;
            elem->np = create_side_path(m, ev->next, pass_ev, verdict);
            return(elem);
        }
        if ((ev->sort == internal) && (ev != pass_ev))
        {
            /* if event is an internal possibility */
```
elem = create_term_path(m, ev->tp, verdict);
if (elem == NULL)
{
    elem = create_empty_element();
    elem->verdict = verdict;
}
else
{
    elem->np = create_side_path(m, ev->next, pass_ev, verdict);
    return (elem);
}

return (elem);

/**************************************************************************/
/* Create a path for verification of a state. The function will search a */
/* unique output event. If it can't find one it will take the first half */
/* unique event and search a unique event from there. This algorithm can */
/* be improved in a next version of this tool. */
/**************************************************************************/
/* TEMPORARY VERSION */
/*****************************/

struct ttcn_el_type *create_verification_path(m, state)
struct efsm_type *m;
struct ext_state_type *state;
{
    struct ttcn_el_type *elem;
    struct event_type *event;
    elem = create_empty_element();
    event = state->first;
    while (event != NULL)
    {
        if (event->unique == yes)
        {
            elem->ev = event;
            elem->nl = create_term_path(m, event->tp, pass);
            if (elem->nl != NULL)
                elem->nl->np =
                    create_side_path(m, event->tp->first, event->tp->first, pass);
            return (elem);
        }
        event = event->next;
    }
    event = state->first;
    while (event != NULL)
    {
        if (event->unique == half)
        {
            elem->ev = event;
            elem->nl = create_term_path(m, event->tp, pass);
            if (elem->nl != NULL)
                elem->nl->np =
                    create_side_path(m, event->tp->first, event->tp->first, pass);
            return (elem);
        }
        event = event->next;
    }
}
elem->nl = create_verification_path(m, event->tp);
elem->np = create_side_path(m, event->fp->first, event, inconclusive);
return(elem);
}
event = event->next;
}
elem = create_term_path(m, state, inconclusive);
return(elem);
}
***************************************************************************/
/* create ttcn reception path possible for every unstable state. It will */
/* wait for the expected input and depending on the next state concatenate */
/* the verification paths. It */
/* returns with a pointer to the first element of this path. */
/***************************************************************************/
struct ttcn_el_type *create_reception_path(m, ev)
struct efsm_type *m;
struct event_type *ev;
{
  struct ttcn_el_type *elem, *el2;
  if (ev != NULL)
  {
    if (ev->sort == output)
    {
      /* if event is an other possibility */
      elem = create_empty_element();
      elem->ev = ev;
      elem->nl = create_verification_path(m, ev->tp);
      elem->np = create_reception_path(m, ev->next);
      return(elem);
    }
    if (ev->sort == internal)
    {
      /* if event is an internal possibility */
      elem = create_verification_path(m, ev->tp);
      elem->np = create_reception_path(m, ev->next);
      return(elem);
    }
    elem = create_reception_path(m, ev->next);
  }
  else
  {
    elem = create_empty_element();
    strcpy(elem->cont, "?OTHERWISE");
    elem->verdict = fail;
    elem->np = create_empty_element();
    el2 = elem->np;
    strcpy(el2->cont, "?EXPIRATION OF TIMER");
    el2->verdict = fail;
  }
  return(elem);
}
***************************************************************************/
/* create ttcn evaluation path possible for every output action. It will */
/* emit the output and depending on the next state concatenate the */
/* verification paths. It */
/* return with a pointer to the first element of this path, with */
/* final verdict verdict */
struct ttcn_el_type *create_eval_path(m, event)
struct efsm_type *m;
struct event_type *event;
{
    struct ttcn_el_type *elem, *nelem;
    elem = create_empty_element();
    elem->ev = event;
    elem->np = create_side_path(m, event->fp->first, event, inconclusive);
    if (event->tp->sort == instable)
    {
        elem->nl = create_reception_path(m, event->tp->first);
        return (elem);
    }
    else
    {
        elem->nl = create_empty_element();
        nelem = elem->nl;
        strcpy(nelem->cont, "START TIMER");
        nelem->nl = create_empty_element();
        nelem = nelem->nl;
        strcpy(nelem->cont, "TIME OUT");
        nelem->nl = create_verification_path(m, event->tp);
        while (nelem->np != NULL) nelem = nelem->np;
        nelem->np = create_empty_element();
        nelem = nelem->np;
        strcpy(nelem->cont, "OTHERWISE");
        nelem->verdict = fail;
        return (elem);
    }
}

/*******************************~******************************~*************/

/* create ttcn path to test an event of machine m */
/* return with a pointer to this path */
/*******************************~******************************~*************/

struct ttcn_path_type *create_path(m, event)
struct efsm_type *m;
struct event_type *event;
{
    struct ttcn_el_type *elem, *pelem; /* next element */
    struct ttcn_path_type *path;
    path = (struct ttcn_path_type *)malloc(sizeof(struct ttcn_path_type));
    sprintf(path->name, " (%s) - %s -> (%s)", event->fp->state->name,
            event->event, event->tp->state->name);
    path->next = NULL;
    elem = create_init_path(m, event->fp);
    if (elem != NULL)
    {
        path->fel = elem;
        while (elem->nl != NULL)
        {
            elem->np =
                create_side_path(m, elem->ev->fp->first, elem->ev, inconclusive);
            elem = elem->nl;
        }
        elem->np =
            create_side_path(m, elem->ev->fp->first, elem->ev, inconclusive);
        elem->nl = create_eval_path(m, event);
struct ttcn_path_type *gen_test_suite(m)
struct efsm_type *m;
{
struct ext_state_type *st;
struct event_type *ev;
struct ttcn_path_type *path, *root;
path = NULL;
st = m->first;
while (st != NULL)
{
ev = st->first;
while (ev != NULL)
{
if (ev->sort == input)
{
if (path == NULL)
{
    path = create_path(m, ev);
    root = path;
}
else
{
    path->next = create_path(m, ev);
    path = path->next;
}
}
ev = ev->next;
}
st = st->next;
}
return(root);
}
Source text of the program Test Suite Generator

```c
/* MODULE OUTPUT.C */
/***************************************************************************/
/* module providing the output of variables created by other modules of */
/* this program. It contains the following 6 functions */
/* write_state, write_event, write_efsm for efsm output */
/* write_test_event, write_elements, write_ttcn_path for ttcn output */
/***************************************************************************/
#include <stdio.h>
#include "typedef.h" /* standard input output functions */
#include *typedef.h* /* definition of used types */

#include <stdio.h> /* standard input output functions */
#include "typedef.h" /* definition of used types */

write_state(out,s)
FILE *out;
struct state_type *s;
{
    fprintf(out,"%s, [%s]", s->name, s->port);
    if (strcmp(s->var,"") != 0) fprintf(out," = \%s")
    else fprintf(out," = \"\"");
}

write_event(out,e)
FILE *out;
struct event_type *e;
{
    fprintf(out,"EVENT : \%d, \%s, \%s\n", e->num, e->event);
    switch (e->unique)
    {
        case yes : fprintf(out,"un, *"); break;
        case half : fprintf(out,"hu, *"); break;
        case no : fprintf(out,"nu, *"); break;
    }
    switch (e->sort)
    {
        case input : fprintf(out,"inp")
        case output : fprintf(out,"out")
        case internal : fprintf(out,"int")
        case syn : fprintf(out,"syn")
        break;
    }
    if (strcmp(e->cond,"") != 0) fprintf(out,"[%s]", e->cond);
    fprintf(out,"\%s \n", e->fp->state->name, e->tp->state->name);
}

write_efsm(out,m)
FILE *out;
struct efsm_type *m;
{
    struct ext_state_type *st;
```
struct event_type *ev;
fprintf(out,"SPECIFICATION OF: ");
write_state(out, m->name);
fprintf(out,"\nWITH FIRST STATE: ");
write_state(out, m->first_state);
fprintf(out,"\nnumber of states: ");
write_state(out, m->num_states);
st = m->first;
while (st != NULL)
{
    fprintf(out,"\nSTATE (%d, ");
    if (st->sort == stable) fprintf(out, "stable, ");
    else fprintf(out, "instable, ");
    write_state(out, st->state);
    fprintf(out,");
    ev = st->first;
    while (ev != NULL)
    {
        write_event(out,ev);
        ev = ev->next;
    }
    st = st->next;
}
fprintf(out,");

/**************************************************************************/
/* write a test event name and if there is a condition the condition on */
/* the screen, with an indentation ind. (later on change ! and ?) */
/**************************************************************************/

write_test_event(out,el,ind)
FILE *out;
struct ttcn_el_type *el;
int ind;
{
    char name[80];
    int i = 0;
    if (el->ev != NULL)
    {
        if (strcmp(el->ev->cond, "!" != 0)
        {
            fprintf(out,"%s",el->ev->cond);
            i = 4 + strlen(el->ev->cond);
        }
        while ((i++) < 15+ind ) fprintf(out," ");
        strcpy(name,el->ev->event);
        if (strchr(name, '!') != NULL) name[1] = '7';
        else if (strchr(name, '!') != NULL) name[1] = '7';
        fprintf(out,"%s",name);
        i += strlen(el->ev->event);
    }
    else
    {
        while ((i++) < 15+ind ) fprintf(out," ");
        fprintf(out,"%s",el->cont);
        i = strlen(el->cont);
    }
    while ((i++) < 60 ) fprintf(out," ");
    switch(el->verdict)
    {
test_suite_generator.c

```c

int case_pass : fprint(out,"pass"); break;
int case_fail  : fprint(out,"fail"); break;
int case_inconclusive : fprint(out,"inconclusive"); break;
} fprint(out,\\n); }

/****************************************************************************
/* write all elements of a TTCN path beginning with el and indentation */
/* this function is a recursive function, that's why it uses this */
/* parameters */
/****************************************************************************

write_elements(out,el,ind)
FILE *out;
struct ttcn_el_type *el;
int ind;
{
   write_test_event(out,el,ind);
   if (el->nl != NULL)
   { write_elements(out,el->nl,ind+2);
   } if (el->np != NULL)
   { write_elements(out,el->np,ind);
   }
}

/****************************************************************************
/* write a TTCN path inclusive comments etc. */
/****************************************************************************

write_ttcn_path(out,p)
FILE *out;
struct ttcn_path_type *p;
{
   struct ttcn_el_type *el;
   int i;
   for(i=0;i<72;i++) fprint(out,"-");
   fprint(out,"\nTTCN PATH NAME: %s\n",p->name);
   for(i=0;i<72;i++) fprint(out,"-");
   fprint(out,"\n\n");
   el = p->fel;
   write_elements(out,el,0);
   fprint(out,"\n");
   for(i=0;i<72;i++) fprint(out,"-");
   fprint(out,"\n\n\n");
}

/****************************************************************************
/* END OF MODULE OUTPUT.C */
/****************************************************************************
```
Using FDT LOTOS to derive tests

`#include <stdio.h> /* standard input output functions */
#include "typedef.h" /* definition of used types */`

`struct efsm_type *read_smile();
struct ttcn_path_type *gen_test_suite();
write_efsm();
write_ttcn_path();`

`main(argc, argv)
int argc;
char *argv[];
{
   FILE *in, *out;
   struct efsm_type *m;
   struct ext_state_type *p;
   struct event_type *e;
   int i=0;
   struct ttcn_path_type *path;
   if ((argc > 3) || (argc < 1))
      {
         printf("use a.out <filename> [<filename>\n"");
         return(0);
      }
   if (argc < 3) out = stdout;
   else
      {
         if ((out = fopen(argv[2],"w")) == NULL)
            {
               printf("opening of file %s failed\n",argv[2]);
               return(0);
            }
         if (argc == 1)
            {
               if ((in = fopen("connect.lot","r")) == NULL)
                  {
                     printf("opening of file connect.lot failed\n");
                     return(0);
                  }
               else if ((in = fopen(argv[1],"r")) == NULL)
                  {
                     printf("opening of file %s failed\n",argv[1]);
                  }
            }
return(0);
}
m = read_smile(in);
fclose(in);
write_efsm(out,m);
path = gen_test_suite(m);
while (path != NULL)
{
    write_ttcn_path(out, path);
    path = path->next;
    if ((i++ % 2) == 1) fprintf(out,"\f");
}
if (argc == 3) fclose(out);

/******************************************/
/* END OF PROGRAM MAIN.C */
/******************************************/