Symbolic Execution of LOTOS Specifications

By
Pierre Ashkar

A M.C.S. thesis submitted
to the school of Graduate Studies & Research
in partial fulfillment of the requirements for the
degree of MASTER of Science in Computer Science
under the auspices of the
Ottawa-Carleton Institute for Computer Science

At the
University of OTTAWA
Ottawa, Ontario, Canada
January 1992

©1992 by P. Ashkar, Ottawa, Canada

Acknowledgments

I am most grateful to my supervisor, Dr. Luigi Logrippo, for all the guidance and valuable advice that he has given me throughout my graduate studies. I also like to thank Dr. Hasan Ural for many useful discussions.

I acknowledge the contributions made by the University of Ottawa LOTOS group; in particular J. Sincennes and M. Haj-Hussein for their help. Special thanks to my girlfriend, Louise, for her patience and support and also, many thanks to my parents.

Abstract

LOTOS (Language Of Temporal Ordering Specification) is a Formal Description Technique (FDT) based on the temporal ordering of observational behaviour. It was developed by ISO (International Organization for Standardization) for the specification of OSI (Open System Interconnection) services and protocols.

This thesis presents a method of translating a LOTOS specification written in any given style into either of two simpler forms, known as the Monolithic Style Specification and the Parameterized Tree.

A method of applying LOTOS expansion theorems to derive an intermediary form, known as the Behaviour Tree, is described first.

A Reduction Algorithm, based on a subset of congruence rules, has been developed to provide a means to eliminate superfluous internal events and branches in the Behaviour Tree prior to conversion to either of the two simpler forms.

Methods of transforming the reduced Behaviour Tree to either the Monolithic Style Specification or Parameterized Tree have been developed.

A tool has been developed to demonstrate these concepts.

To show the application of this tool, we discuss a method for obtaining test cases from a Monolithic Style Specification. Finally, a method to translate such test cases into TTCN form has been outlined.

Contents

Acknowledgments ii
Abstract iii
List of Figures viii
List of Examples ix
Chapter 1 Introduction 1
Section 1 Overview of FDTs 1
Section 2 The University of Ottawa LOTOS Interpreter 2
Section 3 Previous Work 2
Section 4 Motivation of the Thesis 3
Section 5 Objective of the Thesis 3
Section 6 Organization of the Thesis 4
Chapter 2 LOTOS Background 5
Section 1 Overview of LOTOS 5
Topic 1 Introduction 5
Topic 2 LOTOS language 5
Topic 3 Expansions of LOTOS Behaviour Expressions . . . 6
Topic 4 Behaviour Expressions and their Semantics 7
Subtopic 1 Inaction 7
Subtopic 2 Action Prefix 7
Subtopic 3 Internal Action Prefix 8
Subtopic 4 Choice 8
Subtopic 5 Guarded Behaviour 9
Subtopic 6 Process Instantiation 9
Subtopic 7 Local Definition 9
Subtopic 8 Summation on Gates 10
Subtopic 9 Summation on Values 10
Subtopic 10 Parallel Composition 11
Subtopic 11 Par-Expression 12
Subtopic 12 Hiding-Expression 13
Subtopic 13 Successful Termination 14

Subtopic 14 Enable-Expression 14
Subtopic 15 Enable-Expression with Value Passing 14
Subtopic 16 Disable-Expression 15
Topic 5 The Inference System 15
Subtopic 1 Inference Axioms 16
Subtopic 2 Inference Rules 16
Section 2 Specification Styles in LOTOS 19
Topic 1 Introduction 19
Topic 2 The Monolithic Style 19
Topic 3 The Constraint Oriented Style 20
Topic 4 The State Oriented Style 20
Topic 5 The Resource Oriented Style 20
Section 3 Behavioural equivalences 21
Topic 1 Observational equivalence 21
Topic 2 Observation congruence 26
Topic 3 Simplification by congruence rules 27
Section 4 ISLA, the tool at the basis of SELA 28
Topic 1 Introduction 28
Topic 2 Process Menu 30
Topic 3 Process Simulation 30
Chapter 3 Functionality of SELA 33
Section 1 Introduction 33
Topic 1 Definition of a behaviour tree 33
Topic 2 Tasks performed by SELA 34
Topic 3 The use of SELA 35
Subtopic 1 Preparation phase: 36
Subtopic 2 Generation phase: 36
Subtopic 3 Translation phase: 36

Section 2 Expansion 36
Topic 1 Expansion theorems 36
Topic 2 Symbolic expansion 42
Subtopic 1 Explicit Stop or Deadlock 43
Subtopic 2 Duplicate behaviours 44
Subtopic 3 Syntactically equivalent behaviours 45
Subtopic 4 Initial and Terminal Tree Shape Control 47
Section 3 Transformation to monolithic style by using the symbolic expander 49
Section 4 Translation to parameterized tree 52
Chapter 4 SELA Design and Implementation 54
Section 1 Overall Structure 54
Section 2 Internal Form Representation 55
Topic 1 Process Definitions 56
Topic 2 Renaming Data Base 57
Topic 3 Sort Equations and Rewriting Rules 57
Section 3 Intermediate Symbolic Tree Generation 58
Topic 1 Detection of Syntactic Behaviour Equivalences . . 59
Topic 2 Guards and Selection predicates 63
Topic 3 Synchronization 66
Section 4 Generation of Symbolic tree, Parameterized tree and Monolithic specification 68
Topic 1 Symbolic Tree Internal Representation 69
Subtopic 1 Behaviours 70
Subtopic 2 Edges 71
Subtopic 3 Definitions and Instantiations 73
Topic 2 Derivation of the parameterized tree 75
Topic 3 Derivation of LOTOS Monolithic Specification . . . 78
Section 5 Symbolic Tree Reduction 82

Chapter 5 Application of our Work 89
Section 1 Conformance Testing 89
Section 2 LOTOS Trees and Test Sequence Generation 89
Section 3 Discussion of TTCN 90
Topic 1 Test suite overview 91
Topic 2 Declaration part 91
Topic 3 Dynamic part 91
Topic 4 Constraint Part 93
Section 4 Translation of LOTOS Trees to TTCN 93
Topic 1 Definitions 94
Topic 2 Behaviour Expression 95
Section 5 Test Sequence Generation 96
Topic 1 Initialization Step 97
Topic 2 Evaluation Step 99
Topic 3 Termination Step 100
Topic 4 The complete test sequence 100
Topic 5 Test cases in TTCN 102
Chapter 6 Conclusions and Suggestions for Future Work 105
Section 1 Summary of the Thesis 105
Section 2 Limitations 105
Section 3 Future Work 106
Appendix A Case study : Datalink Service Provider 108
Bibliography 125

List of Figures

Figure 1	Behaviour trees of observationally equivalent behaviours	23
Figure 2	Graphical Representation of Specification buffer3	30
Figure 3	Symbolic Expansion Phases	35
Figure 4	Graphical representation of behaviour B	39
Figure 5	Graphical Representation of Behaviour of Ex.23	42
Figure 6	Structure of the Interpreter	55
Figure 7	Graphical representation of the synchronization of a 3–buffers specification	60
Figure 8	Diagram of intermediate symbolic tree	63
Figure 9	Diagram of the monolithic style specification (ex. 31)	86
Figure 10	Test Sequence Derivation	94
Figure 11	Example of a test case structure in TTCN	97
Figure 12	Diagram of specification *buffer3_mono*	98
Figure 13	Diagram of specification *Datalink*	112
Figure 14	Diagram of specification *Datalink*	118

List of Examples

Example 1	Action Prefix	8
Example 2	Choice	8
Example 3	Process Instantiation	9
Example 4	Local definition	10
Example 5	Summation on Gates	10
Example 6	Summation on Values	11
Example 7	Parallel Composition	11
Example 8	Parallel Composition	12
Example 9	Parallel Composition	12
Example 10	Par-Expression	13
Example 11	Hiding-Expression	13
Example 12	Successful Termination	14
Example 13	Enable-Expression	14
Example 14	Enable-Expression with Value Passing	15
Example 15	Disable-Expression	15
Example 16	Bisimulation	22
Example 17	Not Observationally equivalent	25
Example 18	Simplification by congruence rules	28
Example 19	Specification Buffer3	29
Example 20	Expansion Theorem	38
Example 21	Expansion Theorem 2	40
Example 22	Expansion Theorem 3	41
Example 23	Expansion Theorem 4	41
Example 24	Explicit Stop or Deadlock	43
Example 25	Duplicate behaviours	44
Example 26	Syntactically equivalent behaviours	46
Example 27	Upper bounds	48
Example 28	Specification Maximum	50
Example 29	Transformation to monolithic style	51
Example 30	Translation to parameterized tree	53
Example 31	Specification buffer3	61
Example 32	Buffer3 intermediate symbolic tree	62
Example 33	Intermediate Symbolic Tree Generation	65
Example 34	Intermediate Symbolic Tree Generation	67

Example 35 Buffer3 symbolic tree 69
Example 36 Parameterized trees for our examples 76
Example 37 Specification buffer3_mono 79
Example 38 Symbolic Tree Reduction 83
Example 39 Divergence 106

Chapter 1 Introduction

This chapter puts in perspective the context of this thesis. In the first section, an overview of FDTs is given. The FDTs allow precise specifications of distributed systems. They also provide the necessary basis for systematic and (semi-)automatic validation and verification methodologies. The next section briefly describes the LOTOS interpreter which provides the functionalities needed by SELA. Then, a summary of similar work is discussed, followed by the motivation, the objectives and the organization of the thesis.

Section 1 Overview of FDTs

The main objectives of the FDT's (Formal Description Techniques) are to allow the production of OSI (Open Systems Interconnection) standards specifications that are unambiguous, precise and complete and to provide a formally well-defined basis for verification and validation as well as for the conformance testing of implementations [BB87].

The formalization of protocol specifications is important, if not essential, for improving productivity in the process of conformance test suite design and development. Other applications of formalized specifications are:

- Possibility of automating or semi-automating the generation of test cases.
- Validation of specifications.
- Validation of test suites against the formalized specification.
- Possibility of generating an implementation from a formal specification.

Initially, specification of protocols was based mostly on Finite State Machines (FSM) methods. Then, languages based on Extended Finite State Machines (EFSM), like ESTELLE, have appeared. More recently, LOTOS has been proposed as a language for specifying OSI protocols and services.

LOTOS has a different philosophy from EFSM-based languages. In our view, these are useful for describing reference implementations of OSI standards, while LOTOS, because of its abstractness, allows one to produce implementation-independent specifications. The strong mathematical basis of LOTOS allows it to have a precise formal semantics which makes it an executable language.

Section 2 The University of Ottawa LOTOS Interpreter

The Protocols Research Group of the University of Ottawa has developed various *LOTOS-Based* interpreters over the last five years. The current interpreter is based on the LOTOS standard [HH88].

First, the LOTOS specification source is checked for its syntax and static semantics according to [Int88], and, if it is found to be correct, an equivalent 'internal' Prolog form is generated. These functions are written in C and constitute the "LOTOS translator".

The real LOTOS interpreter runs on the internal representation of the data and control components. It consists of two interpreters: The first one is the ADT Interpreter which validates and evaluates value expressions. This interpreter is called SVELDA (System for Validating and Executing LOTOS Data Abstractions) [Feh87]. The second one is the Behaviour Interpreter called ISLA (Interactive System for LOTOS Applications), which helps *to prototype* the dynamic behaviour of LOTOS specifications. These interpreters are programmed in Prolog under the Unix operating system. Currently, a number of other tools are being added to the interpreter, which is evolving towards a "LOTOS Toolkit".

Section 3 Previous Work

The current LOTOS interpreter [HH88] is able to systematically generate *Labelled Symbolic Trees* for a given process up to given maximum lengths and widths. Such trees show all possible execution sequences for the entity specified. When the maximum specified length along a path is exceeded, this is indicated by closing the path with a '*continue*'. Paths exceeding the specified width, are simply truncated with the message "there are other choices".

Further work has been done by Renaud Guillemot [GL89], to improve the presentation of the symbolic tree by:

1. Detecting duplicate behaviour.
2. Detecting and removing some of the non-significant internal events.
3. Eliminating certain paths that are not feasible by trying to evaluate symbolically guards and predicates. Predicates that can be evaluated to false or are in contradiction with others previously assumed to be true, are assumed to be false.

The LOLA (LOtos LAboratory) system developed at Madrid University by J. Quemada, S. Pavon and A. Fernandez [QFP88], is a transformational tool to serve as an experimental environment for a transformational approach for LOTOS. It has been designed in Pascal and it supports the following transformations:

1. Expansion of the specification
2. Parameterized expansion
3. Removal of internal action loops.

The main applications of these transformations are in validating specifications and in implementation derivation. The parameterized expansion of a specification can be seen as a way of producing an efficient implementation or also as a way of transforming constraint oriented specifications into monolithic ones.

Section 4 Motivation of the Thesis

Currently, standard test suites for protocols are derived manually from informal or semi-formal specifications. Unfortunately, this process suffers by the imprecision of the informal or semi-formal notation, and incorrect test cases can be generated. Methods to generate test cases from formal descriptions are being developed in order to overcome this problem [Int92].

Considering the time and effort spent by experts in order to obtain standard test suites [Int89], it would be appropriate if the test suites were systematically derived from formal specifications. The automatic or semi-automatic test suites generation from protocols specified in an FDT such as LOTOS would contribute to the validation process. The need for human effort would be reduced, and would be limited to choosing the most revealing test suites.

The validation of complex LOTOS specifications would be simplified by a method leading to the generation of all possible sequences of actions. Each sequence could then be analyzed and compared to the intents of the designer.

Section 5 Objective of the Thesis

The main objective of this thesis is to present a method of translating a LOTOS specification into simpler forms, suitable for test case derivation. Regardless of

the style of the original specification, the result of the transformations shall be one of the following:

1. Monolitic Style Specification
2. Parameterized Tree

To accomplish this task, the specification is expanded. By applying the LOTOS expansion theorems, all LOTOS expressions can be translated into equivalent expressions using only a small subset of operators, namely the *action prefix* and the *choice*. This form (the Behaviour Tree) is suitable for further transformation.

The Behaviour Tree thus generated might contain superfluous internal events and branches. Based on a set of congruence rules, an algorithm to remove some internal events and branches is developed. The Reduced Behaviour Tree can then be transformed to either of the two simpler forms.

Methods for transforming the Behaviour Tree (reduced or not) to either the Monolithic Style Specification or Parameterized Tree are developed.

In order to demonstrate these concepts, a tool was designed and implemented.

Finally, a method of extracting and converting test cases into TTCN is given.

Section 6 Organization of the Thesis

The thesis is structured as follows. Chapter 2 gives an overview of the LOTOS language and the interpreter ISLA (Interactive System for LOTOS Applications). In Chapter 3, we discuss the features of our tool SELA (Symbolic Execution of LOTOS Applications). In Chapter 4, we discuss the implementation of SELA, and we demonstrate some applications. In Chapter 5, we present a discussion of TTCN (Tree and Tabular Combined Notation) and test sequence generation. The conclusions of the thesis along with a discussion of possible future work follow in Chapter 6. Appendix A presents an application of our tool (SELA).

Chapter 2 LOTOS Background

This chapter is a tutorial of LOTOS history, theory and existing tools on which SELA is based. First, an overview of the language is presented. Then, the specification styles are discussed. This is needed in order to understand one of the goals of SELA, namely the generation of monolithic style specifications. An introduction of the behavioural equivalences follows. The congruence rules presented in this section are used by SELA for reduction purposes. Finally, the LOTOS interpreter (ISLA) and some of its functionnalities used by SELA are described.

Section 1 Overview of LOTOS

Introduction

LOTOS is an FDT (Formal Description Technique) developed within ISO (International Organization for Standardization) in the early eighties, and which now is an ISO standard [Int88], for formally specifying protocols and services of the OSI (Open Systems Interconnection). However, it is generally applicable to other types of distributed, and concurrent systems.

The LOTOS language consists of two main components. The first component, based on ACT-ONE abstract data type formalism, describes the *data* part, while the second component, based on Milner's CCS (Calculus of Communicating Systems) [Mil80] and on Hoare's CSP (Communicating Sequential Processes) [Hoa85], describes the *control* part.

Since LOTOS is (at least partially) an executable specification language, tools which allow to check the static and dynamic semantics of LOTOS specifications can be implemented.

LOTOS language

The *data* component of LOTOS, using the abstract data type language ACT-ONE, describes the data structure manipulated by the control component. An abstract data type is an implementation independent representation of structured data, it is a combination of sorts, operations and equations. The sorts are names of sets of elements, operations are functions on the elements, and equations describe

axioms of the type. The data component will not be discussed in this thesis. However, we use the concept of "value expression"[1].

The *control* component is a combination of processes using LOTOS operators. A process is an entity able to perform internal unobservable actions, and to interact with its environment by means of interactions via common interaction points called *gates*. The atomic form of interaction is an event. An event is a unit of synchronization that may exist between two or more processes that can perform that event.

A LOTOS specification has the form:

specification spec-name [g_1, ... , g_n](v_1, ... , v_m): functionality
behaviour
  <behaviour expression>
where
  <process definitions>
endspec

A process definition has the form:

process proc-name [g_1, ... , g_n](v_1, ... , v_m) : functionality:=
  <behaviour expression>
where
  <process definitions>
endproc

where

spec-name	is the specification name,
proc-name	is the process name,
g_1, ... , g_n	is the list of gates,
v_1, ... , v_m	is the list of formal parameters.

Expansions of LOTOS Behaviour Expressions

In LOTOS, the definition of parallelism is based on the concept of interleaving. More specifically, the fact that actions "a" and "b" can be executed in parallel, is interpreted in LOTOS as allowing execution of "a" followed by execution of "b", or execution of "b" followed by execution of "a".

Milner's *expansion* theorem represents the formalization of this concept. It states that any CCS behaviour expression can be written as a sum of CCS behaviour expressions, where each expression is written as an action followed

[1] A value expression is a term describing a data value, which may contain variables and algebraic operators.

by a behaviour expression. In other words, by applying the *expansion* theorem recursively, it is possible to progressively confine operators different from the sum operator to increasingly more internal sub-expressions. If the given behaviour expression does not contain recursive calls, this will eventually yield to an expression that only contains the sum operator. The *expansion* theorem applies to LOTOS [Int88], if we take the "sum" operator to be the "choice" operator. As we shall see in chapter 3, the *expansion* theorem forms the theoretical basis for our work.

In the following, we explain the LOTOS operators other than "choice" by reducing expressions containing these operators to expressions containing only choices. This provides an appropriate introduction to the concepts in chapter 3.

Behaviour Expressions and their Semantics

In this section, we explain the semantics of the LOTOS operators used to construct the behaviour expressions. The semantic rules, called *inference rules*, are used to derive, from an initial behaviour expression " B ", the set of all possible actions " a " and their resulting behaviours " B' ", meaning that process " B " can execute action " a " and transform into " B' ". As mentioned above, the examples where operators other than action prefix and choice are used are expanded.

Inaction B = stop

The behaviour B cannot offer any action to the environment nor can it perform internal events. *Stop* is equivalent to *DEADLOCK*, however deadlock will not occur if other possibilities exist.

Action Prefix B = a E_1 ... E_n [P] ; B_2

The behaviour B offers participation in event "a E_1 ... E_n [P]"; if the interaction occurs, the resulting behaviour is B_2. "a" is a gate name, and E_i can be either !V, denoting the offering of the value denoted by "value expression" V, or ?x:s, denoting that a value for the variable x of sort s is expected. P is an optional predicate list which should be satisfied in order for B to participate in the event. The E_i are said to be "*value offers*", and [P] is called "*selection predicates*".

Note that *value offers* and *selection predicates* can be absent.

Example 1 Action Prefix

```
process in-out[in,out]:=
   in?x:Nat [x>0 and x<6] ;
   out!x ;
   stop
endproc
```

This process accepts a natural number between `0` and `6` at gate `in`, offers the same value at gate `out`, and then stops.

Internal Action Prefix B = **i** ; B_1

The behaviour B performs an unobservable *internal* action, denoted by **i**, and transforms into B_1.

Choice B = B_1 **[ ]** B_2

The behaviour B may behave either as B_1 or as B_2. The choice may depend on the action offered by the environment.

Example 2 Choice

```
process in1_in2[in1,in2,out]:=
   in1?x:Nat ;
   out!x ;
   stop
  []
   in2?x:Nat ;
   out!x ;
   stop
endproc
```

This process is ready to receive any natural number at gate `in1` or at gate `in2`. It offers the same value at gate `out`, and then stops.

Guarded Behaviour B = **[**Guard**]** —> B_1

The behaviour B will behave as B_1 if the "Guard" is evaluated to true, and behaves as stop otherwise.

Process Instantiation B = P[g_1, ... , g_n](f_1, ... , f_m)

The behaviour B behaves as the behaviour of the process definition P, with the substitution of its formal gates and parameters by the actual gates g_1, ... , g_n and the actual parameters f_1, ... , f_m respectively.

Example 3 Process Instantiation

```
B = in1_in2[a,b,c]
```

where process in1_in2 is as defined above.

The expansion of behaviour B is:

```
   a?x:Nat ; c!x ; stop
[]
   b?x:Nat ; c!x ; stop
```

In LOTOS, infinite behaviours can be defined using recursive *Process Instantiation.*

Local Definition B = **let** x_1=t_1, ... , x_n=t_n **in** B_1

B behaves as B_1 with all occurrences of value identifiers[2] x_1, ... , x_n substituted by value expressions t_1, ... , t_n respectively.

---

[2] A value identifier is a name which can be bound to a data value.

Example 4 Local definition

```
B = let x:Nat=0 , y:Nat=Succ(0) in
      a!x ; b!y ; stop
```

The expansion of this example is:

```
a!0 ; b!Succ(0) ; stop
```

Summation on Gates B = **choice** g **in** [g_1, ... , g_n] [ ] B_1

B behaves as $B_1[g/g_1]$ [ ] ... [ ] $B_1[g/g_n]$, where $[g/g_i]$ denotes that every occurrence of gate g is substituted by gate g_i.

Example 5 Summation on Gates

```
B = Choice a in [in1,in2] []
   a?x:Nat ; out!x ; stop
```

The expansion of this example is:

```
 in1?x:Nat ; out!x ; stop
[]
 in2?x:Nat ; out!x ; stop
```

Summation on Values B = **choice** x:s [ ] B_1

B behaves as $B_1[x/t_1]$ [ ] ... [ ] $B_1[x/t_n]$, where t_1, ... , t_n are all possible value expressions of sort s.

Example 6 Summation on Values

```
B = choice x:Nat [] a!x ; stop
```

The expansion of this example is:

```
  a!0 ; stop
[]
  a!Succ(0) ; stop
[]
  a!Succ(Succ(0)) ; stop
[]
.
.
.
```

Parallel Composition $B = B_1 \; |[g_1, \ldots, g_n]| \; B_2$

The behaviours B_1 and B_2 will behave (offer actions) independently, except for the actions which occur on any of the gates $g_1, \ldots, g_n$, where these two behaviours should synchronize.

The behaviour of the *interleave* operator (|||) is as described above, with an empty set of synchronization gates.

The behaviour of the *full synchronization* operator (||) is as described above, with the set of synchronization gates being the union of all unhidden gates of both behaviours B_1 and B_2.

Example 7 Parallel Composition

```
B = (a;b;d;stop) |[b,d]| (c;b;f;d;stop)
```

The expansion of this example is:

```
  a; c; b; f; d; stop
[]
  c; a; b; f; d; stop
```

Example 8 Parallel Composition

```
B = (a; b; stop) ||| (c; stop)
```

The expansion of this example is :

```
  a; b; c; stop
[]
  a; c; b; stop
[]
  c; a; b; stop
```

Example 9 Parallel Composition

```
B = (a; b; c; stop) || (a; b; d; c; stop)
```

The expansion of this example is:

```
  a; b; stop
```

Note that the composed processes could not execute to termination, since they do not agree on what should follow the first two actions.

Par-Expression B = **par** g **in** [g_1, ... , g_n] *op* B_1

B behaves as $B_1[g/g_1]$ *op* ... *op* $B_1[g/g_n]$, where *op* is a parallel composition operator.

Example 10 Par-Expression

```
B = par g in [g1,g2,g3] || g; f; stop
```

This is equivalent to:

```
  g1; f; stop
||
  g2; f; stop
||
  g3; f; stop
```

Hiding-Expression B = **hide** g_1, ... , g_n **in** B_1

B behaves as B_1, however, any action occurring on any of the gates g_1, ... , g_n is transformed into an internal action. That is, the environment of B_1 cannot participate in these actions.

Example 11 Hiding-Expression

```
B = hide a,b in
            (a;b;d;stop) |[b,d]| (c;b;f;d;stop)
```

The expansion of this example is:

```
  i; c; i; f; d; stop
[]
  c; i; i; f; d; stop
```

Successful Termination B = **exit**(V_1, ... , V_n)

Denotes the successful termination of the behaviour B, where V_1, ... , V_n are instances of **exit** parameters. The resulting behaviour of B after the **exit**, is **stop**.

Example 12 Successful Termination

```
B = g?x:Nat; exit(x)
```

The behaviour expression B accepts a value for `x` at gate g, then it offers `x` at a specific gate called δ and stops.

Enable-Expression B = B_1 >> B_2

B behaves as B_1 until the latter terminates successfully, then B will behave as B_2.

Example 13 Enable-Expression

```
B = (a; b; c; stop [] d; e; exit) >> (f; stop)
```

Note that, in this case, the action at gate δ becomes an internal event.

The expansion of this example is:

```
  a; b; c; stop
[]
  d; e; i; f; stop
```

Enable-Expression with Value Passing B = B_1 >> **accept** x_1:s_1, ... , x_n:s_n in B_2

B behaves as B_1 until it terminates successfully, then B will behave as B_2 with the values exited by B_1 replaced for x_1, ... , x_n.

Note that, in this case also, the action at gate δ becomes an internal event.

Example 14 Enable-Expression with Value Passing

```
B = a?cont:Bool; exit(cont)
      >> accept answer:Bool in (b!answer; stop)
```

The expansion of this example is:

```
a?cont:Bool; i; b!cont; stop
```

Disable-Expression $B = B_1$ **[>** B_2

B behaves as B_1, however during the execution of B_1, B_2 can disable B_1 at any time and start executing. If B_1 is successfully terminated, B_2 can no longer occur.

Example 15 Disable-Expression

```
B = (a; b; c; stop) [> (e; f; stop)
```

The expansion of this example is:

```
  e; f; stop
[]
  a; e; f; stop
[]
  a; b; e; f; stop
[]
  a; b; c; e; f; stop
```

The Inference System

In this section we describe the semantics of a Behaviour Tree in terms of axioms and inference rules.

The notation *infer(B_1, a, B_2)* denotes that B_1 can offer action *a* and then behaves as B_2.

Inference Axioms The following rules are to be taken as axioms:

1. Action Prefix
   `infer(a;B, a, B)`

2. Successful Termination
   a. Without Value Passing
      `infer(exit,`δ` , stop)`
   b. With value Passing
      `infer(exit(`E_1`, ... , `E_n`),`δ E_1`, ... , `E_n`, stop)`

3. Summation on Values
   `infer(choice x:s [] B', dummy?x:s, B')`

The last rule describes the fact that in the case of choice, the user is asked to provide a value.

Inference Rules Inference rules are needed to derive the actions that may be performed by the other behaviour expression constructs.

1. Local Definition

$$\frac{infer([t_1/x_1, ..., t_n/x_n]\, B', a, B'')}{infer(let\ x_1 : s_1 = t_1, ..., x_n : s_n = t_n\ in\ B', a, B'')}$$

2. Guard

$$\frac{eval(P)\ =\ true, infer(B', a, B'')}{infer([P]\ \rightarrow\ B', a, B'')}$$

3. Choice

$$\frac{infer(B_i, a, B_i')}{infer(B_1[]B_2[]...[]B_n, a, B_i')}$$

   where $B_i\ \epsilon\ \{\ B_1\ ,\ B_2\ ,\ ...\ ,\ B_n\}$

4. Hiding

$$\frac{infer(B', a, B''), name(a)\ \notin\ \{g_1, ..., g_n\}}{infer(hide\, g_1, ..., g_n\, in\, B', a, hide\, g_1, ..., g_n\, in\, B'')}$$

$$\frac{infer(B', a, B''), name(a) \in \{g_1, ..., g_n\}}{infer(hide\, g_1, ..., g_n\, in\, B', i(a), hide\ g_1, ..., g_n\ in\ B'')}$$

5. Nested

$$\frac{infer(B, a, B')}{infer((B), a, B')}$$

6. Parallel

   a. Selected Synchronization

$$\frac{\begin{array}{c} infer(B_1, a_1, B_1'), infer(B_2, a_2, B_2'), \\ name(a_1) = name(a_2) \in \{g_1, ..., g_n, \delta\} \end{array}}{infer(B_1|[g_1, ..., g_n]|B_2, a_3, B_1'|[g_1, ..., g_n]|B_2')}$$

   where a_3 is the resulting interaction by matching a_1 and a_2.

$$\frac{infer(B_1, a, B_1'), name(a) \notin \{g_1, ..., g_n, \delta\}}{infer(B_1|[g_1, ..., g_n]|B_2, a, B_1'|[g_1, ..., g_n]|B_2)}$$

$$\frac{infer(B_2, a, B_2'), name(a) \notin \{g_1, ..., g_n, \delta\}}{infer(B_1|[g_1, ..., g_n]|B_2, a, B_1|[g_1, ..., g_n]|B_2')}$$

   b. Interleaving

$$\frac{infer(B_1 |[]| B_2, a, B')}{infer(B_1 ||| B_2, a, B')}$$

c. Full Synchronization

$$\frac{infer(B_1\,|[g_1,...,g_n]|\,B_2,\,a,\,B')}{infer(B_1\,||\,B_2,\,a,\,B')}$$

where {g1, ... , gn} = G,, is the union of all possible gates of B1 and B2. In this case the second and third inference rules of selected synchronization can never be applied.

7. Enable

$$\frac{infer(B_1,\,a,\,B_1'),\,name(a)\;\neq\;\delta.}{infer(B_1\;>>\;accept\,x_1:s_1,...,x_n:s_n\,in\,B_2,\,a,\;B_1'\;>>\;accept\,x_1:s_1,...,x_n:s_n\,in\,B_2)}$$

$$\frac{infer(B_1,\,\delta E_1,...,E_n,\,B_1')}{infer(B_1\;>>\;accept\,x_1:s_1,...,x_n:s_n in\,B_2,\,i,\;[E_1/x_1,...,E_n/x_n]\,B_2)}$$

In this case B_1' is equal to stop.

8. Disable

$$\frac{infer(B_1,\,a,\,B_1'),\,name(a)\;\neq\;\delta}{infer(B_1\,[>\;B_2,\,a,\,B_1'\,[>\;B_2)}$$

$$\frac{infer(B_1,\,\delta E_1,...,E_n,\,B_1')}{infer(B_1\,[>\;B_2,\,\delta E_1,...,E_n,\,B_1')}$$

In this case also B_1' is equal to stop.

$$\frac{infer(B_2,\,a,\,B_2')}{infer(B_1\,[>\;B_2,\,a,\,B_2')}$$

9. Summation on Gates

$$\frac{infer((B')[g_i/g],\, a,\, B'')}{infer(choice\ g\ in\ [g_1, ..., g_n]\ []\ B',\, a,\, B'')}$$

is an inference rule for each $g_i \in \{g_1, \ldots, g_n\}$.

10. Par-Expression

$$\frac{infer(B'[g_1/g]\ op\ ...\ op\ B'\ [g_n/g],\, a,\, B'')}{infer(par\ g\ in\ [g_1, ..., g]\ op\ B',\, a,\, B'')}$$

is an inference rule where $g, g_1, \ldots, g_n$ are gate-variable instances, op is a parallel-operator.

11. Process Instantiation

$$\frac{infer(([t_1/x_1, ..., t_m/x_m][g_1/h_1, ..., g_n/h_n],\, a,\, B''B))}{infer(p[g_1, ..., g_n](t_1, ..., t_m),\, a,\, B')}$$

is an inference rule **iff** there exists a process definition:
$p[h_1, ..., h_n](x_1 : s_1, ..., x_m : s_m) \ := \ B$

12. Stop

There are no inference rules for stop because it cannot generate any action.

Section 2 Specification Styles in LOTOS

Introduction

The development of formal specifications of OSI service and protocol architectures is a difficult task. One of the most important problems is to find an appropriate structure for the specification. Different specification styles can be used. A brief discussion of various styles in designing LOTOS specifications follows. For additional details, see [VSvS88].

The Monolithic Style

In the monolithic style, all possible sequences of actions are shown explicitly, and the specification is represented as one part, without distinguishing between local aspects in the observational behaviour.

This style is most useful for the design of simple specifications, but it is of little practical use for more complex specifications. Because of the lack of structure, it results in specifications of large size, which can be difficult for human understanding.

Since the only constructs allowed are sequential composition, choice, guards and process instantiation (a call to a process), specifications structured in this style are close to transition trees. Therefore, they can be very useful for the derivation of test cases.

The Constraint Oriented Style

The constraint oriented style is very suitable for specifying the abstract, implementation independent behaviour of systems in a modular fashion. This allows to express different aspects of a complex system in separate, manageable and well structured specification modules.

However, since this style induces extensive usage of parallel composition operators and parameterization, it may be hard to grasp the overall behaviour of a specification from a knowledge of its components and their relationships. This style relies on the powerful, but hard to implement, LOTOS features for multi-way synchronization on multi-valued event offers.

The State Oriented Style

In this style, the system is regarded as a single resource whose internal state space is explicitly defined as a collection of alternative sequences of observable interactions. Each alternative is guarded by a test to determine whether the system is in a given state.

This style is particularly useful in the end-phase, where the specification can be seen as an object that can be implemented by a single implementation module.

Like the monolithic style, this style does not result in well structured specifications. Therefore, it is not suitable where abstractness is needed.

The Resource Oriented Style

The resource oriented style supports the description of the system in terms of modules representing different resources, which interact among themselves

through interfaces. Each resource module can be assigned to a single implementation module, whose functionality is specified with straightforward mappings onto implementation constructs.

In the resource oriented style, each resource can be specified using a constraint oriented, state oriented or a monolithic approach.

Section 3 Behavioural equivalences

There are two types of actions, *observable* (external) actions, in which the environment can participate, and the *unobservable* (internal) action denoted by *i*, which is hidden from the environment.

Observational equivalence

Two behaviours are considered *observationally equivalent* if, for any sequence of externally observable actions, the sequence is accepted (rejected) by one behaviour, iff it is accepted (rejected) by the other, and application of the sequence to both behaviours leads to observationally equivalent behaviours[BB87].

Definition Observable Sequence Relation

Let B, B' and B_i denote tree nodes.

1. Let s denote a string $x_1\, x_2 \ldots x_n$ of actions (not necessarily observable). We define relation $-s->$ as the obvious extension of the transition relation to action sequences: $B-s-> B'$ iff there exist $B_i, 0 \leq i \leq n$, such that $B = B_0 - x_1 -> B_1 \ldots B_{n-1} - x_n -> B_n = B'$. In particular, for $n = 0$ we have $B - \varepsilon -> B$ for any B, where ε is the empty string.
2. Let s denote now a string $a_1\, a_2 \ldots a_n$ of *observable actions*, and let i^k denote a sequence of $k (k \geq 0)$ $i-$actions. Then we have an *observable sequence relation* noted $B = s => B'$ whenever there exists a sequence $\left(i^{k0}\, a_1\, i^{k1}\, a_2 \ldots a_n\, i^{kn}\right)$ of actions such that $B - \left(i^{k0}\, a_1\, i^{k1}\, a_2 \ldots a_n\, i^{kn}\right) -> B'$. This implies that $B = \varepsilon => B'$ whenever $B - i^k -> B'$, for $k \geq 0$, and that for any B, $B = \varepsilon => B$ (in this case $k = 0$).

The purpose of this relation is to abstract from the unobservable actions that are on the path between two tree nodes.

Based on the observable sequence relation, we define a notion of *bisimulation*[BB87].

Definition Observational Equivalence

A binary relation $\Re$ between tree nodes is a *Weak Bisimulation* if for any pair (B1 , B2) in $\Re$ and for any string S of observable actions:

1— Whenever $B_1 = s => B_1'$, then, for some B_2':

$$B_2 = s => B_2' \; and \; B_1' \; \Re \; B_2'$$

2— Whenever $B_2 = s => B_2'$, then, for some B_1':

$$B_1 = s => B_1' \; and \; B_1' \; \Re \; B_2'$$

The idea of *weak bisimulation* is that two bisimilar nodes must be able to simulate each other, in terms of observable sequences, and then reach still bisimilar nodes.

Two tree nodes B_1 and B_2 are *observationally equivalent* (noted $B_1 \approx B_2$), if there exists a *weak bisimulation* $\Re$ which contains the pair (B_1, B_2).

Example 16 Bisimulation

The following is an example of two observationally equivalent behaviours B_1 and B_2:

$$B_1 \; : \; a; (b; stop \; [\,] \; i; c; stop) \; [\,] \; a; c; stop$$
$$B_2 \; : \; a; (b; stop [\,] i; c; stop)$$

The corresponding behaviour trees are shown in figure 1, where the dotted lines show the *weak bisimulation* relation that can be established between the nodes of the two trees.

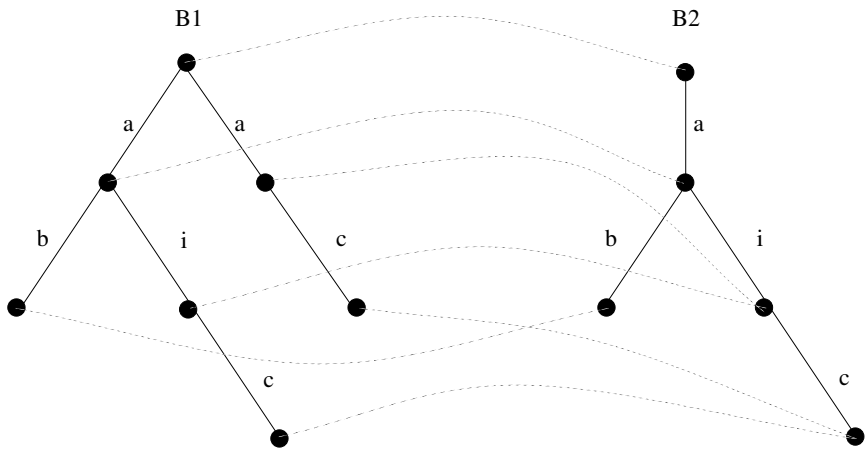
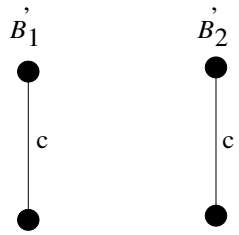


Figure 1 Behaviour trees of observationally equivalent behaviours

Let s be the observable action a.

Now, $B_1 = a => B_1'$ but also by performing $a\,i$, $B_2 = a => B_2'$ where



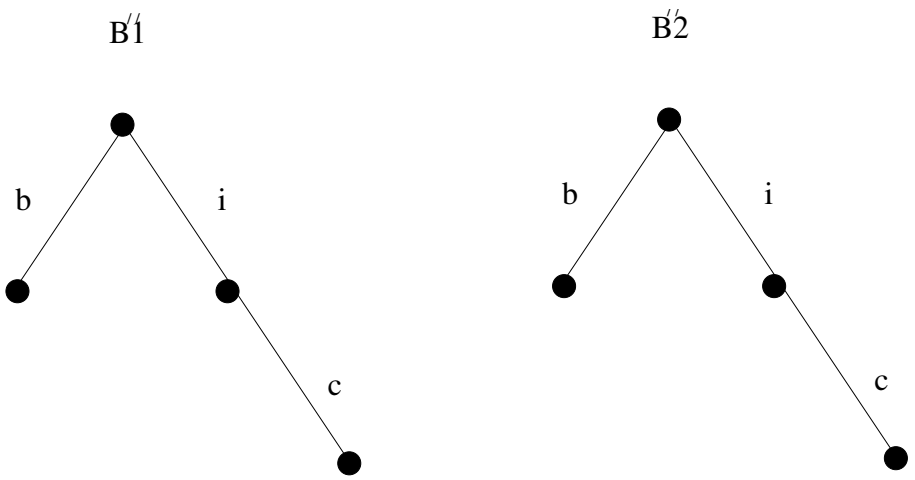
B_1' and B_2' are observationally equivalent.

Let s' be the observable action a.

By performing a on both behaviours B_1 and B_2:

$$B_1 = s' => B_1''$$
$$B_2 = s' => B_2''$$

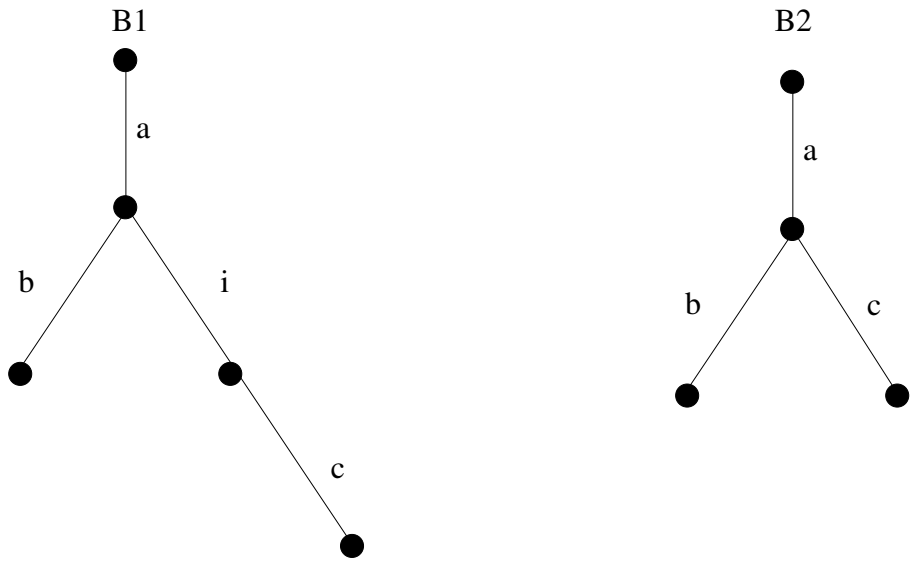
B_1'' and B_2'' are also observationally equivalent:



After having checked all other possible sequences of actions in a similar way, we can conclude that B_1 and B_2 are in fact observationally equivalent.

Example 17 Not Observationally equivalent

The following is an example of two behaviours B_1 and B_2 which are not observationally equivalent:



Let s be the observable action a.

$$B_1 \quad = \quad s \quad => \quad B_1'$$

however also

$$B_2 \quad = \quad s \quad => \quad B_2'$$

where

B_2' can perform the actions b or c, but B_1' can only perform the action c, therefore, B_1' and B_2' are not observationally equivalent.

Observation congruence

Two behaviours B_1 and B_2 are said to be *observationally congruent* (noted $B_1 \overset{c}{\approx} B_2$), if by replacing B_1 by B_2 in any context, the result of the replacement will be observationally equivalent to the original.

Definition

$B_1 \overset{c}{\approx} B_2$ (Observation congruence) iff for every expression context $C[]$, $C[B_1] \approx C[B_2]$. Three observational congruence rules are stated:

$$1 - \ a;i;B \overset{c}{\approx} a;B$$
$$2 - \ B\,[\,]i;B \overset{c}{\approx} i;B$$
$$3 - \ a;\,(B[\,]B) \overset{c}{\approx} a;B$$

where a denotes any action.

The following is an example of two observationally congruent behaviours B_1 and B_2:

$$B_1 \ : \ a;(b;stop[\,]i;b;stop)$$
$$B_2 \ : \ a;b;stop$$

By applying the second rule, B_1 can be reduced to:

$$B_1 \ : \ a;i;b;stop$$

By applying the first rule, B_1 can be further reduced to:

$$B_1 \ : \ a; b; stop$$

Which means that $B_1 \overset{c}{\approx} B_2$. (Note that in LOTOS there is an infinite number of such Congruence rules. Many are mentioned in [Int88]).

The observational congruence relation is stronger than the observational equivalence relation. The following example shows two observationally equivalent ($\approx$) behaviours B_1 and B_2, which are not observationally congruent ($\overset{c}{\approx}$).

$$B_1 \ : \ i; stop$$
$$B_2 \ : \ stop$$

$B_1 \ \approx \ B_2$ since $(i; stop \ , \ stop)$ is a bisimulation, however

$$B_1 \overset{c}{\not\approx} B_2$$

Let $B_3 = i; stop[\,]a; stop$, if we replace B_1 in B_3 by B_2 we get $B_3' = stop[\,]a; stop$. Now by performing the internal action i (the empty sequence ϵ) on B_3

$$B_3 \ = \ \varepsilon \ => \ B_{3'}$$

where $B_{3'} \ : \ stop$. However, the empty sequence cannot lead to $stop$ in B_3'. Therefore B_3 and B_3' are not observationally equivalent.

Simplification by congruence rules

Congruence rules can be used to reduce behaviour trees by removing some internal events and even some branches. Currently, only *congruence rules* 1), 2) and 3) mentioned above are used in our system. It is possible, however, to extend the system to include additional rules.

Consider, for example, the following behaviour expression:

Example 18 Simplification by congruence rules

```
B = a ; ( c ; d ; stop
            []
          i ; c ; d ; stop )
      []
    a ; ( i ; c ; d ; stop
            []
          c ; i ; d ; stop )
```

By applying congruence rules 1 and 2, B will be simplified to:

```
B = a ; c ; d ; stop
      []
    a ; c ; d ; stop
```

By applying rule 3, the expression can be further reduced to:

```
B = a ; c ; d ; stop
```

Section 4 ISLA, the tool at the basis of SELA

Introduction

LOTOS has been designed as an executable specification language even if executability was not the primary goal. An interpreter called ISLA (Interactive System for LOTOS Applications) has been implemented at the University of Ottawa, that supports many functions to simulate the execution of a specification and debug it (a similar interpreter has been developed in Europe in the framework of an ESPRIT project [vE89]). The advantage of having ISLA is to validate the actual behaviour of a specification with respect to its intended behaviour.

In the following we discuss the functions of interest; for more details consult [HH88].

In order to help the reader in understanding the functions discussed in this section, an example taken from [QFP88] is developed to show the results of every function. This specification inputs one `bit` through the gate `in1` and outputs the same `bit` through the gate `out`. The specification is composed by synchronizing

over the hidden gates `m1` and `m2` three instantiations of process buffer1. Process buffer1 inputs a `bit` through gate `in1` and outputs the same value through gate `out`. Once the value is exited through the gate `out` then the process buffer1 is ready to receive a new `bit` through the gate `in1`. The following figure illustrates the behaviour of the specification and the synchronization between all the processes.

Example 19 Specification Buffer3

```
 1  SPECIFICATION buffer3 [ in1 , out ] : noexit
 2
 3  BEHAVIOUR
 4       hide m1,m2 in
 5            ( (  buffer1 [ in1 ,m1 ]
 6               |[ m1 ]|
 7                 buffer1 [ m1 , m2 ]
 8               )
 9               |[ m2 ]|
10               buffer1 [ m2 , out ]
11             )
12
13
14  WHERE
15     PROCESS buffer1 [  in1 , out  ] : noexit :=
16              in1 ?x:bit ;
17              out !x ;
18              buffer1 [ in1 , out ]
19     ENDPROC
20
21  ENDTYPE
22  ENDSPEC
```

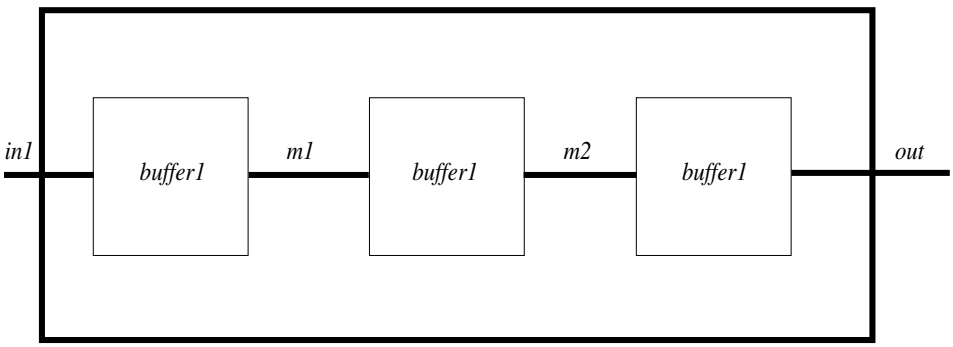


Figure 2 Graphical Representation of Specification **buffer3**

Process Menu

Once a LOTOS specification is compiled and translated into internal form representation, it can be loaded for execution. On completion, a menu of its process headers is listed. Each header consists of process name, formal gates, formal parameters (if any) and a unique number starting with the number of the main specification, which is usually *1*. The following is the process menu of the `buffer3` example given above:

```
The available processes are:
[1] buffer3[in1,out]()
[2] buffer1[in1,out]()
PROC: lp[rocesses]|?|h[elp]|<process number>|<command>
==>
```

To execute a process, its number should be entered, followed by the list of actual value parameters (if any).

Process Simulation

During the simulation, at each step a menu of all possible actions and their resulting behaviour expressions is given. Since a behaviour expression can be large, unique indexed symbols are used to represent them in the menu.

An action in the menu has the following form:

<*i*> — *action* — — —> bh_i [LN_1 , ... , LN_m]

where

i

is the number associated with the given action (index)

action

is the action executed.

bhi

is a symbol representing the resulting behaviour if action i is chosen

$[LN_1, ..., LN_m]$

is a list of line numbers of the interacting actions (*action*$_1$, ... , *action*$_m$) in the source specification

The following is an action menu taken from the `buffer3` specification, once the number `1` in the process menu has been chosen which corresponds to `buffer3`.

```
=====================================================
  <1>- in1 ?x:bit  ---> bh1  [16]
=====================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command>
==> 1
 Enter a value for x:bit => 1
=====================================================
```

The only action offered is on gate `in1`. Once the number which corresponds to this action is chosen, a value of type `bit` is requested from the user.

Subsequently, the following action is offered:

```
=====================================================
  <1>- i (hiding: m1 !1)  ---> bh1  [17,16]
=====================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command>
==> 1
   Internal event is executed
 Passed evaluated value ==> 1
```

The bit received through gate `in1` is transferred to the second instantiation of process `buffer1` through the hidden gate `m1`.

The following two actions are then offered:

```
=====================================================
  <1>- in1 ?x:bit  ---> bh1  [16]
  <2>- i (hiding: m2 !1)  ---> bh2  [17,16]
=====================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command>
==>
```

Any one of these two actions can be chosen. In the first action offered, a bit must be entered through gate `in1` since the first instantiation of `buffer1` is empty. The second action is the result of the synchronization over the hidden gate `m2`, where the value received through gate `m1` is ready to be sent through gate `m2` to the third instantiation of process `buffer1`.

Chapter 3 Functionality of SELA

In this chapter, we discuss SELA from the point of view of features offered to the user, and from the point of view of LOTOS theory. In this section, we give and introduction of SELA; its basic operations and organization. A tutorial of the expansion theorems is given next. It serves as an introduction to the symbolic expansion. The last sections cover the final transformations of the generated symbolic tree, namely, the monolithic style specification and the parameterized tree.

Section 1 Introduction

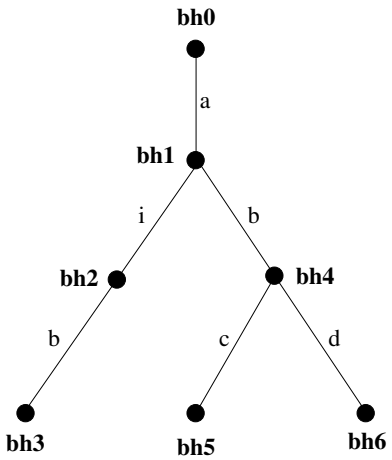
Definition of a behaviour tree

The dynamic behaviour of a LOTOS specification can be represented by a tree, called behaviour tree, where the nodes of the tree represent the states of the behaviour, and the arcs represent the actions on which the behaviour may be derived.

For example, given the following LOTOS behaviour:

$$a;(i;b;stop[\,]b;(c;stop[\,]d;stop))$$

The corresponding behaviour tree is:



where

bh0 = original behaviour

bh1 = (i ; b ; stop [] b ; (c ; stop [] d ; stop))

bh2 = b ; stop

bh4 = (c ; stop [] d ; stop)

bh3 = bh5 = bh6 = stop

Two nodes are said to be:

- Duplicate, if they are identical as chains of characters,
- Syntactically equivalent, if they are duplicate with respect to the difference in their associated value expressions.

Tasks performed by SELA

The purpose of SELA is to generate the behaviour tree of a specification (or a process) symbolically, that is, without the use of actual values.

The basic operation of SELA is as follows. From the initial behaviour, a symbolic behaviour tree is generated. Since such a tree may have a large depth and width (sometimes infinite), the user is expected to specify these boundaries. While generating the symbolic behaviour tree, each node reached is recorded and checked against the ones previously generated in order to detect duplicates and syntactic equivalences between them.

From there on, SELA supports the translation to parameterized tree, where the first occurrence of all nodes having a duplicate or a syntactically equivalent, are transformed to an independent subtree having a unique name and a list of some formal parameters, which makes the tree simpler to follow.

SELA also provides the means to translate the generated tree into a monolithic style specification and parameterized tree useful for in-depth analysis of the original specification. Similarly as in the translation to a parameterized tree, duplicate and syntactically equivalent nodes are transformed to an independent process.

The use of SELA

The symbolic expansion process is divided into the three phases depicted in figure 3. The shaded areas represent the functionality of SELA. A method of generating TTCN test cases from either representation is discussed in chapter 5.

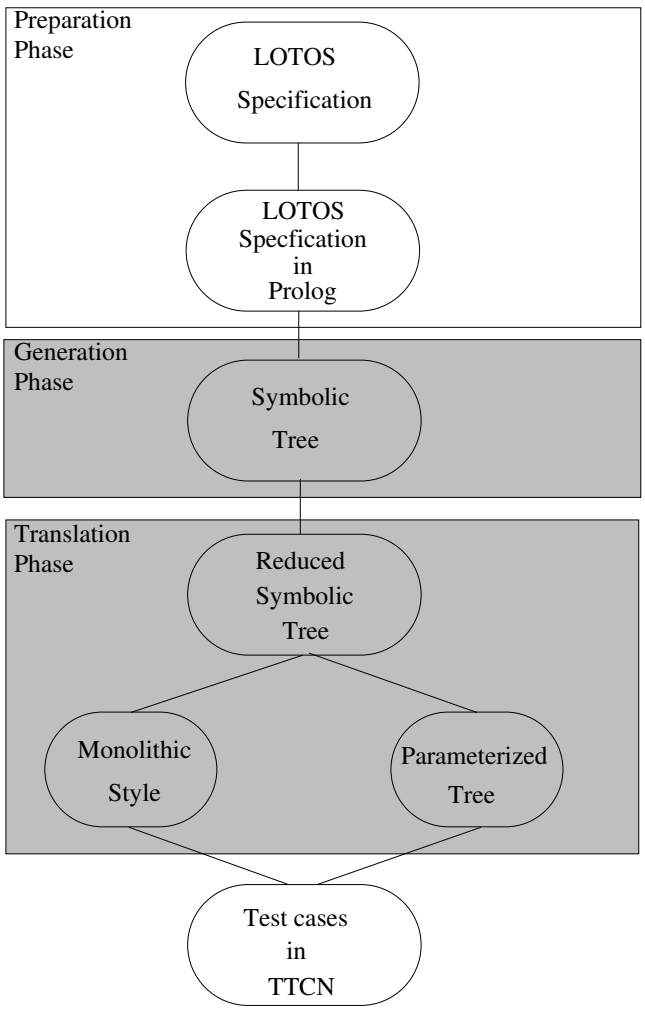


Figure 3 Symbolic Expansion Phases

Preparation phase: preparing the proper execution environment consists of:

1— using the LOTOS Translator to translate the LOTOS source specification into its corresponding Prolog internal representation, suitable for use with ISLA and SELA.
2— invoking the ISLA Simulator, to load the specification in internal representation.

Generation phase: the generation phase deals with the generation of the symbolic tree from the Prolog representation of the specification. The tree is created by SELA according to the boundaries, namely the depth and the width, specified by the user. It is stored in a Prolog database.

Translation phase: the translation phase corresponds to the generation of the output desired by the specifier for analysis. This can be in one of the following formats:

1. The reduction of the symbolic tree by application of congruence rules.
2. The translation of the symbolic tree into a monolithic style specification.
3. The translation of the symbolic tree into a parameterized tree.

Section 2 Expansion

Expansion theorems

In chapter 2, when we presented the various LOTOS operators, we showed how expressions containing them could be *expanded* into expressions containing only the "choice" operator. In this section, we present the theoretical justification for this transformation, which are LOTOS congruence rules called *expansion theorems*. They are the application to LOTOS of Milner's theorems by the same name [Mil80].

In order to express the expansion theorems, we introduce some notation and conventions. First, we generalize the choice operator [] to accept an arbitrary number of arguments. We shall write $[]S$, where S is a set of behaviour expressions which can be enumerated by some suitably chosen index set $\{B_1, B_2, \ldots, B_n\}$, to denote the behaviour expressions $B_1[]B_2[]...[]B_n$

Furthermore, we use the fact that an action-prefix including " ?" can always be translated into a choice of action-prefixes only containing " !" by using the following law:

$$g...\ ?x\ :\ t...\ ;\ B\ = \textbf{choice}\ x\ :\ t[]\ g...\ \ !x...\ ;\ B$$

where x is a value identifier of type t, and B is a behaviour expression.

Finally, we assume that the elements of S are all of the form $b_i\ ;\ B_i$ where b_i denotes any action and B_i the resulting behaviour.

Let $B\ =\ []\{b_i; B_i \mid i \in I\}\ ,\ C\ =\ []\{c_j;\ C_j \mid j \in J\}$

where I and J denote the indexed sets of B and C respectively.

1

$$B|[A]|\ C\ \overset{c}{\approx}$$
$$[]\{b_i\ ;\ B_i\ |[A]|\ C\ \mid\ name(b_i)\ \notin A,\ i \in I\}$$
$$[][]\{c_j\ ;\ (B\ |[A]|C_j\ \mid\ name(c_j) \notin A,\ j \in J\ )\}$$
$$[][]\{a; (B_i\ |[A]|\ C_j\ \mid\ a = b_i = c_j\ ,\ name(a)\ \in A, i \in I, j \in J)\}$$

if all $b_i(i \in I), c_j(j \in J)$ are of the form $g!E_1 ... !E_n$

where $name\,(b_i)$ denotes the gate used in the action b_i and A is a list of gate-identifiers where the two behaviours B and C interact.

Example 20 Expansion Theorem

Let B = B1 $|[g]|$ B2 where `B1` and `B2` are executed independently, except for the actions occurring on gate `g`, where `B1` and `B2` must synchronize. Furthermore, let

```
B1 = (g!succ(0); g1!succ(0); stop
      []
      g1?y:nat; g!y; stop
     )
B2 = (g ?x:nat; g2!x; stop
      []
      g2?z:nat; g!z; stop
      )
```

By applying the above expansion theorem, `B` is congruent to:

```
B = (g1?y:nat; (g2?z:nat; [y=z]-> g!y; stop
                []
                g!y; g2!y; stop
               )
     []
     g2?z:nat; (g1?y:nat; [z=y]-> g!y; stop
                []
               [z=succ(0)]->g!succ(0);g1!succ(0);stop
               )
     []
     g!succ(0); (g1!succ(0); g2!succ(0); stop
                 []
                 g2!succ(0); g1!succ(0); stop
                 )
      )
```

As mentioned above, an action of the form `g1?y:nat` is to be read as a shorthand for an infinite **choice**. In order to represent the synchronization between the behaviour expressions `B1` and `B2` on gate `g`, a **guard** on their values offered on the named gate is added.

Figure 4 shows the graphical representation of the behaviour `B` given above.

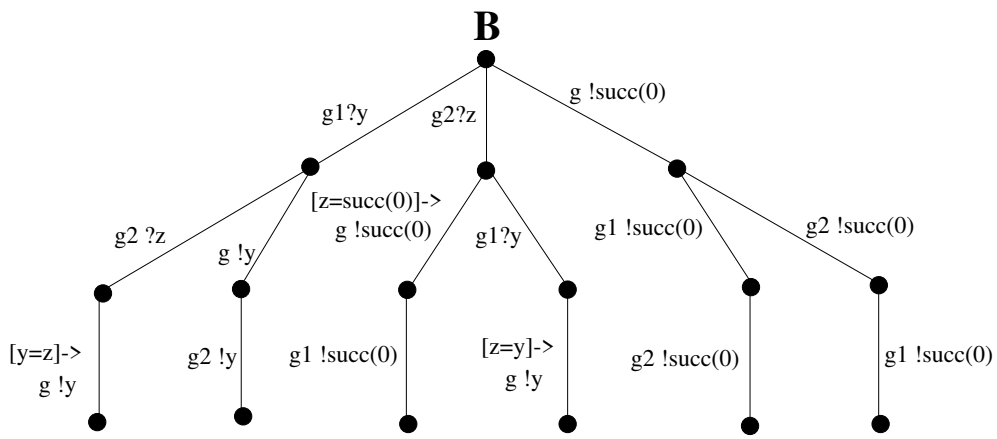


Figure 4 Graphical representation of behaviour B

2

$$B \,[>\; C \;\overset{c}{\approx}\; C$$

$$[][]\{b_i \,;\, (B_i \,[>\; C) \mid i \;\in\; I\}$$

Example 21 Expansion Theorem 2

Let `B = B1 [> B2` where

```
B1 = g?x:nat; g!x; stop

B2 = a?y:nat; a!y; stop
```

By applying this expansion theorem, `B` is congruent to:

```
B = (g?x:nat; (g!x; a?y:nat; a!y; stop
               []
               a ?y:nat; a !y; stop
               )
     []
     a ?y:nat; a !y; stop
     )
```

`B1` can offer two successive actions on gate `g`, unless `B2` interacts on gate `a`, in which case it will disable `B1` (i.e. `B2` can disable `B1` at any time).

3 $hide\ A\ in\ B \overset{c}{\approx}$

$$[]\{b_i\ ;\ hide\ A\ in\ B_i \mid name(b_i) \notin A, i \in I\}$$
$$[][]\ \{i\ ;hide\ A\ in\ B_i \mid name(b_i) \in A, i \in I\}$$

if all $b_i(i \in I)$ are of the form $g!E_1 ... !E_n$

Example 22 Expansion Theorem 3

Let `B = hide a in B1` where

```
B1 =  ( a?x:nat; B1
        []
         b?y:nat; b!y; B1
        )
```

By applying this expansion theorem,

```
B = (i; B
     []
      b?y:nat; b!y; B
     )
```

The action performed on gate `a` becomes an internal action, and is replaced by `i`, whereas the actions performed on gate `b` are left as external actions.

4

$$B\,[S] \overset{c}{\approx} \; []\,\{S(b_i)\,;\, B_i\,[S] \mid i \in I\}$$

Example 23 Expansion Theorem 4

Let

```
B = (a?x:nat; a!x; stop
     []
     b?x:nat; B
    )
```

Note that this behaviour is already expanded. Its graphical representation is shown in figure 5.

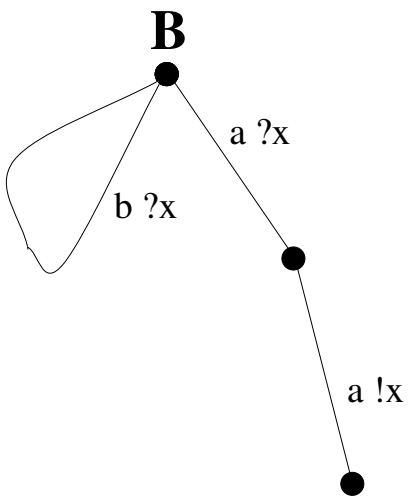


Figure 5 Graphical Representation of Behaviour of Ex.23

Note that after action `b` the resulting behaviour is the original behaviour `B` itself.

Symbolic expansion

The symbolic expander SELA uses the inference rules of the interpreter (ISLA) to expand the symbolic behaviour tree of a given specification, process or a current behaviour.

By applying LOTOS inference rules to a behaviour expression B, the set of actions a and the set of resulting behaviour expressions B' can be found, such that $B - a- > B'$. By iteratively applying the same rules on the resulting behaviour expressions B' the behaviour tree of B can be generated. A label 'bh n' where 'n' is a unique integer number starting with the value `0` is assigned to each behaviour, and it is incremented by `1` for every new generated behaviour.

The expansion of each branch ends when one of the following states is reached:

1– Explicit Stop or Deadlock.

2– Duplicate or syntactically equivalent behaviours.

3– One of the upper bounds (depth or width).

Explicit Stop or Deadlock The exploration of a branch will end whenever a *stop* action in the original behaviour is executed, or a deadlock results from synchronization. In either case, we have a “DEADLOCK”.

Example 24 Explicit Stop or Deadlock

```
 1  specification Basic_lotos[a,b,c,d,e,f]:noexit
 2
 3   behaviour
 4      ( a ; b; (c; d; stop
 5                  []
 6                 e; stop))
 7           ||
 8      ( a; b; (c; f; stop
 9                  []
10                 e; stop))
11
12  endspec
```

The corresponding behaviour tree is:

```
bh0 * 1 a [4,8]
bh1 * | 1 b [4,8]
bh2 * | | 1 c [4,8] DEADLOCK
    * | | 2 e [6,10] DEADLOCK
```

Where the first ’DEADLOCK’ is the result of impossible synchronization between the gates ’d’ and ’f’, appearing on lines 4 and 8 respectively, while the second is the explicit *stop* performed in the original specification.

Duplicate behaviours Repetition can be detected whenever a current behaviour B is found to be *syntactically equal* to a previous one B'. Two behaviours are said to be syntactically equal, iff they are identical as chains of characters. When such repetitions are found, the behaviour B will be presented as `again bh n`, where `n` is the behaviour number of B'. In what follows, a behaviour such as B that has been labeled in this way, will be called a *transfer behaviour*, the corresponding behaviour tree node will be called a *transfer node*, and a behaviour such as B' that has a *transfer behaviour* which refers to it, will be called a *target behaviour*, the corresponding behaviour tree node will be called a *target node*.

Example 25 Duplicate behaviours

```
behaviour
          simple1[Q,A]
where
   process simple[Q,A]:noexit:=
        Q ; A ; simple1[Q,A]
   endproc
```

The corresponding behaviour tree is:

```
bh0 * 1 Q
bh1 * | 1 A
bh2 * | | 1 Q
bh3 * | | | 1 A
  .
  .
  .
```

Note that the sequence of actions `Q`, `A` is repeated since the next action is the instantiation of the process `simple1` itself. Hence `bh0` and `bh2` are duplicates.

The corresponding behaviour tree once the duplicate behaviours are detected is:

```
bh0 * 1 Q
bh1 * | 1 A  ==> again bh0
```

where `bh2` is represented as `again bh0`

Syntactically equivalent behaviours In some cases, behaviours are identical except for the value of some value expressions.

This can be due to recursion, where the current behaviour is the instantiation of a process (or a set of processes in the case of parallelism) which had already been instantiated previously. The detection is achieved by replacing all bound value identifiers (i.e. value identifiers which were bound to some value expressions) appearing in these behaviours by new value identifiers, and comparing the resulting behaviours to see if they are duplicates. This will be explained in greater detail in chapter 4, §3.

This state is presented in the same way as in the duplicate behaviour case `again bh n`.

Example 26 Syntactically equivalent behaviours

```
specification QA_service [A]:noexit
TYPE Nat is
        sorts
                nat
        opns
                0       :       -> nat
                succ    : nat   -> nat
endtype
behaviour
         simple2[A](0)
where
   process simple2[A](q:nat):noexit:=
        A!q ;
                simple2[A](succ(q))
   endproc
endspec
```

The corresponding tree is:

```
bh0 * 1 A !0
bh1 * | 1 A !succ(0)
bh2 * | | 1 A !succ(succ(0))
  .
  .
  .
```

The process `simple2` offers the value of `q` through the gate `A` and re-instantiates itself while incrementing the value identifier `q` by one. Note that behaviours `bh0` and `bh1` are not duplicates due to the difference of their value expressions, but by disregarding these values, these two behaviours are identicals, and therefore they are considered for our purpose to be *syntactically equivalent*.

The corresponding tree once the syntactically equivalent behaviours are detected is:

```
bh0 * 1 A !0  ==> again bh0
```

Note that the tree notation ignores parameters. They will be shown in another type of representation to be discussed below.

Initial and Terminal Tree Shape Control The generation of the symbolic tree can be initiated in one of two main ways. First of all, the original specification, or any of its processes, may be used as the starting point of the generation. It suffices to provide the name of the desired process or specification. Furthermore, at any time during the *step-by-step* execution of a specification or process (see §4), it is possible to use the current behaviour to start the generation of the symbolic tree.

These different initiation techniques offer flexibility in the generation of the tree. While starting with the specification leads to the generation of the complete symbolic tree, it is also possible to limit the generation of the symbolic tree to one of the behaviours derived from the execution of a predefined sequence of actions.

Since a behaviour tree can be infinite, the user is required to provide values for the boundaries of the tree, namely, the width (the maximum number of next actions for each node) and the depth (maximum number of successive multiple derivations) of the tree.

Therefore, the expansion of a branch stops whenever the width or depth of the tree reaches the maximum values provided by the user. In the case of depth, the node reached is labeled “continue”, while in the case of width, the first N (where N is the width provided by the user) alternatives are generated and the rest are simply truncated with the message “there are other choices”.

Further control could be applied to limit the growth of the tree. For instance, it would be possible to prune the subtrees when:

1. a specific action is met
2. a specific gate is encountered
3. a particular state is reached

The study of such possibilities is left to further work.

Example 27 Upper bounds

```
 1  specification depth[a,b]:noexit
 2
 3   behaviour
 4    p[a,b]
 5  where
 6
 7  process  p[a,b]: noexit:=
 8       a ; p[a,b]
 9       |||
10       b; stop
11  endproc
12  endspec
```

If the maximum depth is specified to be 3, the corresponding tree is:

```
bh0 * 1 a [8]
bh1 * | 1 a [8]
bh2 * | | 1 a [8] ==> continue
    * | | 2 b [10] ==> continue
    * | | 3 b [10] ==> continue
    * | | 4 b [10] ==> continue
    * | 2 b [10]
bh3 * | | 1 a [8] ==> continue
    * | | 2 b [10] ==> continue
    * | 3 b [10]
bh4 * | | 1 a [8] ==> continue
    * | | 2 b [10] ==> continue
    * 2 b [10]
bh5 * | 1 a [8] ==> again bh4
```

Note that the resulting behaviour of `bh5` once the action `a` is executed is found to be a duplicate of behaviour `bh4`, therefore the resulting behaviour was considered as a *transfer node* and was replaced by the label `again bh4`.

Section 3 Transformation to monolithic style by using the symbolic expander

The tree resulting from the expansion process described above is saved and further transformed to a monolithic style specification. Note that all value identifiers bound in syntactically equivalent behaviours, which were replaced by new value identifiers, were saved in a *parameter-list* as pairs. The first element of such a pair will be called *actual parameter* (bounded value identifier) and the second element *formal parameter* (new value identifier) respectively. This is done in order to be able to reinsert such value identifiers in the behaviour expression as parameters during the transformation.

In the process of transformation to a monolithic style specification, four different types of tree nodes have to be considered, each different type being transformed in a unique way:

1— A deadlock node is replaced by a "stop".
2— A *target node* should be interpreted as an independent subtree, therefore it is transformed into a process definition having its related *formal parameters* as a parameter list.
3— A *transfer node* (again bh n), is transformed into a process instantiation of a process "n", and having its related *actual parameters* as a parameter list.
4— External, internal actions and continue are represented in the same way as they appear in the symbolic tree.

Example 28 Specification Maximum

Following is a specification of an entity which takes two natural numbers in any order, outputs the largest of them, and then repeats its behaviour.

```
 1  specification Maximum[in1,in2,out] :noexit
 2
 3  behaviour
 4          Max1[in1,in2,out]
 5  where
 6          process Max1 [in1,in2,out] : noexit :=
 7
 8             Max2 [in1,in2,out]
 9               >> Max1 [in1,in2,out]
10          endproc
11
12          process Max2 [val1,val2,max] : exit :=
13
14              ( val1?X:int; exit(X, any int)
15                |||
16                val2?Y:int; exit(any int, Y))
17
18                 >> accept V: int, W: int in
19                       max!largest(V,W);exit
20          endproc
21   endspec
```

The corresponding tree is:

```
bh0 * 1 in1 ?int@1:int [14]
bh1 * | 1 in2 ?int@2:int [16]
bh2 * | | 1 i (enable: exit !int@1 !int@2) [14,16]
bh3 * | | | 1 out !largest(int@1,int@2) [19]
bh4 * | | | | 1 i (enable: exit) [19] ==> again bh0
    * 2 in2 ?int@1:int [16]
bh5 * | 1 in1 ?int@2:int [14] ==> again bh2
```

where int@'n' is the value identifier bound to a symbolic value 'n' of type int. This will be explained in greater detail in chapter 4, §3.

Example 29 Transformation to monolithic style

Given specification Maximum, from the preceding example. The corresponding monolithic style specification, after applying the congruence rules to remove the internal actions produced by the enable operator in the original specification (see §3), appears as follows:

```
specification maximum_mono [in1,in2,out] : noexit
  behaviour
       P0[in1,in2,out]
  where
process  P0[in1,in2,out]:noexit :=
 in1 ?int@1:int ;
    in2 ?int@2:int ;
       P2[in1,in2,out](int@1,int@2)
 []
 in2 ?int@1:int ;
    in1 ?int@2:int ;
       P2[in1,in2,out](int@2,int@1)
endproc
process  P2[in1,in2,out](var_1, var_2:int):noexit :=
 out !largest(var_1,var_2) ;
    P0[in1,in2,out]
endproc
```

Note that there are two *transfer nodes* `bh4 (again bh0)` and `bh5 (again bh2)` in the symbolic tree above. These two nodes are transformed into process instantiations `P0` and `P2` respectively. Their corresponding *target nodes* `bh0` and `bh2` are transformed into process definitions `P0` and `P2` respectively. As the reader may have noticed the order in which the actions are presented differs between the two forms (`bh2`). This is due to the extraction of the *target nodes* from the original tree, in order to be presented as a process definition in the monolithic specification form.

Section 4 Translation to parameterized tree

The transformation into *parameterized tree* is in some way similar to the transformation into monolithic style. The tree resulting from the expansion process is saved in order for the transformation to be applied.

Parameterized Trees provide a more compact format to convey the same information contained in the monolithic specification. They are similar to the symbolic trees discussed above. However the tree is split in sections, where each section is similar to a process, since it is considered to be capable to be instantiated with different actual parameters. Therefore, we introduce the concept of *subtree heading*, which consists of a subtree name, followed by a formal parameter list.

The only difference between these two transformations is in the treatment of the tree nodes.

In the process of transformation to *parameterized tree*, three different types of tree nodes have to be considered:

1— External and internal actions, and deadlock and continue are represented in the same way as they appear in the behaviour tree.

2— A *transfer node* (`again bh n`) is transformed into a tree instantiation of `tree<n>` followed by its related *actual parameters* as a parameter list.

3— A *target node* (`bh n`) is interpreted as an independent subtree, therefore it is transformed into a subtree by having a subtree heading `tree<n>` (where `n` is the behaviour number) followed by its related *formal parameters* as a parameter list. Considering this fact, in order to hold the path between the subtree (`tree<n>`) and its father node, the *target node* is also replaced by a tree instantiation of the new formed subtree, of the form `tree<n>`, followed by its related *actual parameters* as a parameter list.

Example 30 Translation to parameterized tree

The *parameterized tree* corresponding to the specification shown above, after applying the congruence rules for simplification , is:

```
tree<0>
bh0 * 1 in1 ?int@1:int [14]
bh1 * | 1 in2 ?int@2:int [16] ==>tree<2>(int@1,int@2)
tree<2>(var_1, var_2:int)
bh2 * | | | 1 out !largest(var_1,var_2) [19] ==>tree<0>
    * 2 in2 ?int@1:int [16]
bh3 * | 1 in1 ?int@2:int [14] ==>tree<2>(int@2,int@1)
```

In this example, we have two *target nodes* `bh0` and `bh2`. The node `bh0` is the root of the tree, therefore it is considered automatically as an independent subtree, even if it is not necessarily a *target node*.

Since the node `bh2` is considered as a *target node*, the *parameterized tree* is split at this node by giving the remaining section the *subtree heading* `tree<2>` followed by the formal parameters list `(var_1, var_2:int)`. The two value identifiers which were bound to `int@1` and `int@2` at gates `in1` and `in2` respectively, are passed as a parameter list to the subtree `tree<2>`.

The two *transfer nodes* `again bh0` and `again bh2` are transformed into subtree instantiations `tree<0>` and `tree<2>` respectively. In the case of *transfer node* `again bh2`, the instantiation is followed by the two value identifiers which were bound to `int@2` and `int@1` at gates `in1` and `in2` respectively.

In the case where the main subtree `tree<0>` has a formal parameters list, the *parameterized tree* starts with an instantiation of this subtree, which consists of the main subtree name followed by the actual parameters list.

Note that both *parameterized tree* and monolithic specification are derived directly from the symbolic tree. Because of the way the derivation is done, the structure of the two is usually different as is obvious in our example.

Chapter 4 SELA Design and Implementation

In this chapter, details of the design and the implementation of SELA are presented. To begin with, the general organization of SELA is explained; how it fits with the existing tools, and the reasons for certain implementation choices. The following section presents specifics of the internal form representation; its components and their purposes. Then, the generation of the intermediate symbolic tree is discussed. This includes explanations on detection of behavioural equivalences and a description of problems encountered in the generation of the tree. Section 4 describes the generation of the symbolic tree and its translation to either of monolithic style specification or parameterized tree. The last section discusses the reduction of the symbolic tree using congruence rules.

Section 1 Overall Structure

This section describes the overall structure of SELA (Symbolic Expander for LOTOS Applications) and the implementation of important components of its functionalities.

SELA is programmed in Quintus Prolog under UNIX. Prolog was chosen as the implementation language for the following reasons:

1. The ISLA interpreter, with which SELA interfaces, is programmed in Prolog.
2. Prolog supports dynamic allocations for facts and rules, unification, direct substitution and backtracking. These are all important in the programming of a system based on inference rules.
3. Individual functionalities can be easily added, tested, modified or removed. This allows extensive experimentation to take place at little cost.

In addition, SELA uses the behaviour Interpreter, called ISLA (Interactive Systems for LOTOS Applications [HH88]), and the ADT Interpreter, called SVELDA (System for Validating and Executing LOTOS Data Abstractions) [Feh87].

Both interpreters and SELA are implemented in Quintus Prolog and run on the 'internal' form, which is a Prolog representation of the LOTOS source specification.

Figure 6 shows the general structure of all components used by SELA.

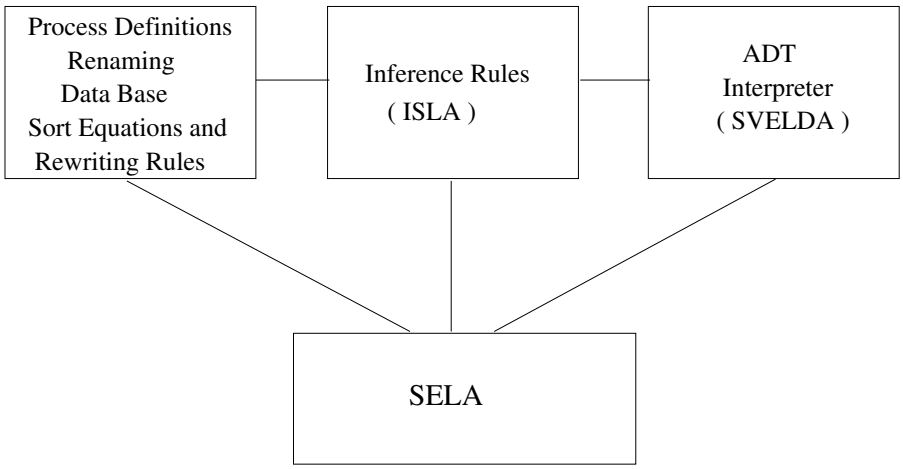


Figure 6 Structure of the Interpreter

Section 2 below describes the LOTOS internal form representation. The inference rules of ISLA where discussed in chapter 2 §1. These were not part of the author's work, however it is necessary to understand them in order to understand SELA's implementation.

Section 2 Internal Form Representation

The 'internal' form representation of a LOTOS specification consists of three major parts:

1— Process Definitions.
2— Renaming Data Base.
3— Sort Equations and Rewriting Rules.

Every identifier in the LOTOS specification is renamed by a unique internal name to prevent ambiguities in the interpretation, since a specification may contain different identifiers with the same external name.

The identifiers refer to: variables, gates, process names, operators, sorts, and types. Their internal form names begin with 'v', 'g', 'p', 'o', 's' and 't' respectively, followed by a unique numerical index.

Process Definitions

Process definitions have the following form:

proc(<proc-id>, [<formal-gates>], (<formal-pars>), <beh-exp>).

where

<proc-id>	is the process name.
<formal-gates>	is the list of the named process' formal gates.
<formal-pars>	is the list of formal parameters, and this list can be empty.
<beh-exp>	is the named process' behavior expression.

A formal parameter has the following form:

[<prolog-var>, <var-name>, <var-sort>]

where

<prolog-var>	is a unique Prolog variable, that holds the value of the matching LOTOS parameter.
<var-name>	is the internal parameter name.
<var-sort>	is the internal parameter sort.

The same <prolog-var> occurs in the behaviour expression, so that when a value is assigned to this parameter, then all occurrences of the same variable will be instantiated to the same given value.

There exist two different types of actions in LOTOS:

Internal actions are represented as:

i([<line-number>])

where

<line_number>	is the line number where this action occurs in the source specification.

External actions are represented as:

[<gate>([<line-number>]), [<experiments>], [<predicate>]]

where

<gate>	is the internal name of the gate, on which this action takes place.
<experiments>	is a list of zero or more experiments.
<predicate>	is the list of the action's selection predicates, if any.

An experiment has one of the following formats:

1— [$, <prolog-var>, <var-name>, <var-sort>]
where

$	represents the 'accept' (?) experiment

2— [#, val(<prolog-var>, <var-name>), <var-sort>]
where

#	represents the 'offer' (!) experiment

Renaming Data Base

As mentioned above, external identifiers are replaced by internal ones. In order to be able to recover the external identifiers to display to the user, a renaming data base is used.

The renaming data base has the following form:

renm(internal-name, 'external-name', [nesting]).

where

'internal-name'	is the internal name of the identifier.
'external-name'	is the external name of the identifier.
[nesting]	is the internal name of the structure where the identifier is defined.

Sort Equations and Rewriting Rules

Sort equations and rewriting rules are the basic components of the ADT interpreter (SVELDA)[Feh87].

In order to evaluate a guard or a selection predicate, SELA calls the ADT interpreter (SVELDA) by sending the value expression to be evaluated. There is no direct use of the sort equations and the rewriting rules by SELA.

Section 3 Intermediate Symbolic Tree Generation

For reasons that are given in §4, the generation of the symbolic tree is done in two steps. The first time, we obtain what we call the *intermediate* symbolic tree. We then proceed to obtain the symbolic tree in its final form. From this one, we obtain the *parameterized tree* and the *monolithic style* specification.

The intermediate symbolic tree is generated by applying the inference rules listed in the previous section. This is done by the Prolog call:

infer(B, A, BN, LN, [], _)

where

B	is the current behaviour.
A	is the generated action.
BN	is the new behaviour resulting from the current behaviour B, once the action A is executed.
LN	is the line number where the action A appears in the source specification.

Since SELA is implemented using Prolog, backtracking facilities help to derive all possible actions A_i and their resulting behaviours BN_i.

Some of the derived actions may contain value identifiers yet to be bound by the environment. Each occurrence of such value identifiers is replaced by a *symbolic value* which consists of:

— The sort of the value identifier,
— A symbol which expresses how the value would be bound, i.e. @ for a value identifier at a gate, and **%** for a value identifier in a *choice*, an *exit(any)*, or an initial process parameter,
— A numerical value ' i ' (i = 1,2, ... ,n), where ' i ' is the i^{th} binding occurrence in the current path.
— If needed, a second numerical value preceded by a dot is added to distinguish between different value identifiers of the same sort and the same nesting level.

The expression *symbolic value* refers to the element resulting of the replacement of an *unbound value identifier* (e.g. *nat@1* as explained above). On the other hand, the expression *value identifier* is a synonym of *bound value identifier*, which is already bound to a value expression.

Detection of Syntactic Behaviour Equivalences

In order to be able to identify *transfer* and *target* nodes in the symbolic trees as they are being generated, it is necessary to detect symbolic equivalence between behaviour expressions. For this to be possible, the internal representation of the behaviour expression must be translated in a more convenient form. We proceed as follows:

1— all gates' internal names are replaced by their external names.
2— all information added in the translation to internal form representation, such as action line numbers, is removed.
3— in case of process instantiation, all parameter values are replaced by new value identifiers.
4— symbolic values and value identifiers are also replaced by new value identifiers.

A symbolic value or a value identifier which appears more than once in a behaviour is replaced by the same new value identifier in all appearances in the behaviour. This criterion reduces the chance of detecting the equivalence between the behaviours, but it also reduces the number of parameters resulting from the cleaning of the behaviour. The above four steps constitute what we call *cleaning* a behaviour expression.

SELA maintains a list (*beh list*) of all behaviour expressions encountered so far in the symbolic evaluation process. Such expressions are in *clean* form. Each expression is identified by a unique number.

Once a behaviour is *cleaned*, it is compared with the behaviours already in the *beh list*. In case of matching, the current behaviour is transformed into a process instantiation, in other words, the current behaviour is replaced by an *again* associated with the existing syntactic equivalent behaviour's number, and the generation of the current path is stopped.

Otherwise, the current behaviour is saved in the *beh list* with an associated unique number, and the generation of the path is carried on.

For illustration, the LOTOS specification shown in chapter 2 will be used for the rest of the chapter, in order to help the reader in understanding the functions discussed in this chapter.

This specification inputs one `bit` through the gate `in1` and outputs the same `bit` through the gate `out`. The specification is composed by synchronizing over

the hidden gates `m1` and `m2` three instantiations of process buffer1. Process buffer1 inputs a `bit` through gate `in1` and outputs the same value through gate `out`. Once the value is exited through gate `out` then the process buffer1 is ready to receive a new `bit` through gate `in1`. The following figure illustrates the behaviour of the specification and the synchronization between all the processes.

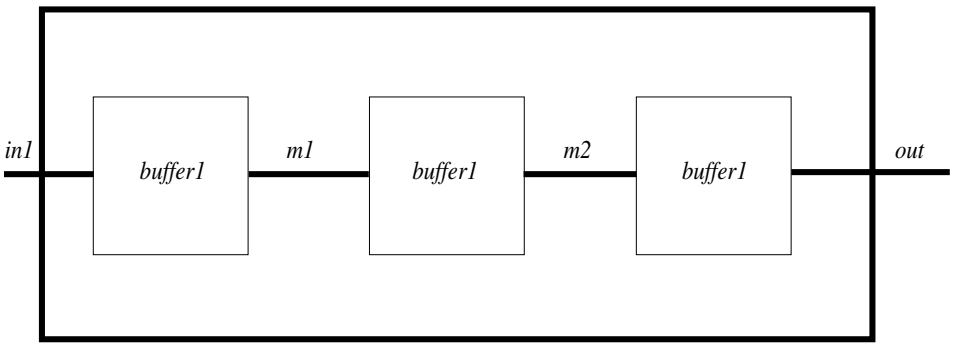


Figure 7 Graphical representation of the synchronization of a 3–buffers specification

Example 31 Specification buffer3

```
 1  SPECIFICATION buffer3 [ in1 , out ] : noexit
 2   (* Example due to Quemada and Pavon *)
 3  BEHAVIOUR
 4       hide m1,m2 in
 5            ( (  buffer1 [ in1 ,m1 ]
 6               |[ m1 ]|
 7                 buffer1 [ m1 , m2 ]
 8               )
 9               |[ m2 ]|
10               buffer1 [ m2 , out ]
11             )
12
13
14  WHERE
15     PROCESS buffer1 [   , out  ] : noexit :=
16              in1 ?x:bit ;
17              out !x ;
18              buffer1 [ in1 , out ]
19     ENDPROC
20
21  TYPE bit1 IS
22     SORTS bit
23     OPNS 0,1  : -> bit
24      inc : bit -> bit
25     EQNS
26     OFSORT bit
27     inc(0) = 1 ;
28     inc(1) = 0 ;
29  ENDTYPE
30  ENDSPEC
```

Example 32 Buffer3 intermediate symbolic tree

The corresponding intermediate symbolic tree is:

```
bh0 * 1 in1 ?bit@1:bit [16]
bh1 * | 1 i (hiding: m1 !bit@1) [17,16]
bh2 * | | 1 in1 ?bit@2:bit [16]
bh3 * | | | 1 i (hiding: m2 !bit@1) [17,16]
bh4 * | | | | 1 i (hiding: m1 !bit@2) [17,16]
bh5 * | | | | | 1 in1 ?bit@3:bit [16]
bh6 * | | | | | | 1 out !bit@1 [17] ==> again bh3
    * | | | | | 2 out !bit@1 [17] ==> again bh2
    * | | | | 2 out !bit@1 [17] ==> again bh1
    * | | 2 i (hiding: m2 !bit@1) [17,16]
bh7 * | | | 1 in1 ?bit@2:bit [16] ==> again bh4
    * | | | 2 out !bit@1 [17] ==> again bh0
```

The behaviours `bh0, bh1, bh2, bh3 and bh4` are called *target* behaviours, since they have *transfer* behaviours (`again`) which refer to them.

Note that, as it was mentioned in chapter 3, the generation of a branch can also end if a *Deadlock* or the maximum depth specified by the user has been reached. In diagram form, this intermediate symbolic tree is shown in figure 8

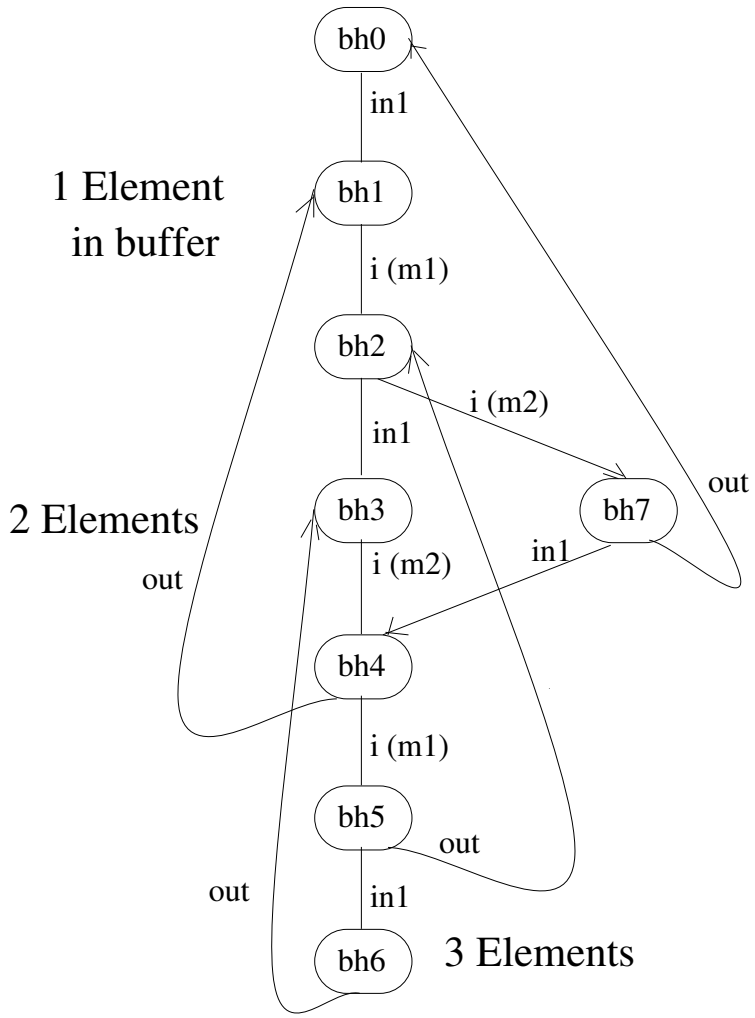


Figure 8 Diagram of intermediate symbolic tree

Guards and Selection predicates

Formal parameters can have different values for different instantiations. Therefore, based on their current values, the evaluation to *false* of guards or

selection predicates is not sufficient to allow the truncation of a branch.

Guards and selection predicates that are evaluated to:

- [**True**] are eliminated from the branch (since they are always true, they are always redundant).
- [**False**] and contain only *ground terms*[3] are eliminated and so is the branch that follows them, since they will always be evaluated to false.
- [**False**] and contain at least one formal parameter or a value identifier (which can have different values for different instantiations) are added to the branch.

The guards and selection predicates which cannot be evaluated due to the presence of symbolic values are also added to the branch.

Consider the following specification that can accept a value expression at gate `z` and passes two values to `process input` which are `succ(z)` and `z`. The `process input` in turn will compare these two values, and based on the evaluation, it will either accept a value expression for `x` or `y` or transmit the value `0`.

[3] The set $TERM(OP, s)$ of *ground terms* of sort $s \in S$ is defined as the set $TERM(OP, \emptyset, s)$, where $TERM(OP, V, s)$ is the set of terms of sort $s \in S$ with operations in OP and variables in the set of variables V, and with <S,OP> being a signature. In other words, ground terms contain no variables.

Example 33 Intermediate Symbolic Tree Generation

```
 1 specification Nat_numbers[g]:exit
 2
 3 behaviour
 4
 5      data[g]
 6        >>  accept x,y:nat in
 7      input[g](x,y)
 8
 9      where
10           process data[g]:exit(nat,nat) :=
11                  g?z:nat; exit(succ(z),z)
12           endproc
13
14           process input[g](x,y:nat) :exit :=
15            [x > y] -> g?x:nat ; input[g](x,succ(y))
16            []
17            [x < y] -> g?y:nat ; input[g](x,y)
18            []
19            [x = y] -> g!0 ; stop
20           endproc
21
22 library Boolean Natural endlib
23
24 endspec
```

Considering the two values passed to `process input` the first *guard* (`[x > y]`) will always be evaluated to *true* the first instantiation, and the last two are *false*. Therefore, one could at first think that only the first branch (`g?x:nat ; input[g](x,succ(y))` should be traced, and the other two branches should simply be truncated. This would not be the desired result, however, because the other two guards could themselves be evaluated to *true* in successive instantiations. Therefore, the symbolic generation cannot be based on the evaluation of the *guards* and *selection predicates* only. It must be based on the type of the values that are involved (*ground terms or value identifiers*) also. The corresponding *intermediate* symbolic tree is:

```
bh0 * 1 g ?nat@1:nat [11]
bh1 * | 1 i (enable: exit !succ(nat@1) !nat@1) [11]
bh2 * | | 1 [succ(nat@1)>nat@1] g ?nat@2:nat [15] ==> again bh2
    * | | 2 [succ(nat@1)<nat@1] g ?nat@2:nat [17] ==> again bh2
    * | | 3 [succ(nat@1)=nat@1] g !0 [19] DEADLOCK
```

Example 33 (Continued) Intermediate Symbolic Tree Generation

Note that our current value expression evaluator is not capable to conclude that "for all naturals `n`, `succ(n) < n` is false", and that therefore 'line 17' leads to a `deadlock`. An enhanced evaluator that is able to detect such contradictions has been developed recently in our group.

Furthermore, even if it was possible to conclude that such a value expression is false, it would still be necessary to keep the branch that depends on it in the tree, because it could end up in a loop (as in fact happens in this case), where the same value identifiers could become bound to different value expressions at different iterations. Therefore, at this stage, the apparently illogical tests that we find in the *intermediate* symbolic tree are simply place-markers due to the fact that the tree is generated top-down, and will be replaced by other tests during further processing, leading to the symbolic tree in its final form.

Synchronization

The *synchronization* is similar to the *guard* and *Selection predicate* case, the matching of values is done in the same way, however the related condition is added as a selection predicate to the resulting node.

This can be illustrated in a LOTOS specification as follows:

Example 34 Intermediate Symbolic Tree Generation

```
 1 specification Nat_numbers[g,g1]:exit
 2
 3 behaviour

 5 where
 6  process p[g,g1](x:nat) :exit :=
 7
 8     ( ( g!succ(0) ; exit
 9         ||
10          g!x ; exit
11        )
12    []
13     g1; exit
14     )
15   >>  p[g,g1](succ(x))
16
17  endproc
18
19 type Nat_numbers is
20
21     sorts nat
22
23     opns 0:        -> nat
24          succ: nat -> nat
25
26 endtype
27 endspec
```

Note that the offers on line 8 and 10 can only synchronize if `x = succ(0)`. In the first instantiation of `process p` on line 4, `x` will have the value `0`, therefore the synchronization would fail. Whereas in the second instantiation of `process p` line 15, `x` will have the value `succ(0)` and the synchronization would succeed and generate the action `g! succ(0)`, therefore this branch should not be pruned.

In other words, we have here a synchronization condition `succ(0) = x`, which, since `x` has the value `0`, evaluates to false. However since `x` could have a different value later, the condition `succ(0) = 0` is still added to the branch as a **selection predicate** to the action. The corresponding *intermediate* symbolic tree:

```
bh0 * 1 g !succ(0)
        [succ(0)=0]  [8,10]
bh1 * | 1 i (enable: exit) [8,10] ==> again bh0
    * 2 g1 [13]
bh2 * | 1 i (enable: exit) [13] ==> again bh0
```

Example 34 (Continued) Intermediate Symbolic Tree Generation

Again, this apparently illogical representation is due to the fact that the tree is generated topdown. Therefore, at the time `succ(0) = 0` is generated, we do not know yet that this condition will end up within a loop in the final symbolic tree. Further processing will eliminate such apparently incongruous terms, by replacing *ground terms* such as `0` and `succ(0)` by new *value identifiers* (this matter is covered later on in section 5). We shall see in the next section how the trees of examples 4.1 and 4.2 are further transformed into parameterized trees and monolithic specifications.

Section 4 Generation of Symbolic tree, Parameterized tree and Monolithic specification

In this section, we attempt to give the reader an idea of the *symbolic tree*'s internal form representation, how it is derived, and how it is used. We avoid many details, which would be extremely tedious to explain completely.

In the process of translation to parameterized tree or to monolithic style specification, all value expressions and actual parameters appearing in a *target* behaviour which will be translated to process definition, should be replaced by new value identifiers (formal parameters) before deriving the subtree of the behaviour. Every such value identifier has the form `var_n`, where n starts with the value '1' and is incremented by one for every new value identifier in the same behaviour.

Since the *target* behaviours cannot be known before actually generating the *symbolic tree* and finding all syntactic equivalent and duplicate behaviours, the tree should be generated twice: in the first generation of the tree, discussed in the previous section, a list of all the numbers associated with the *target* behaviours is saved in the *targeted_list*. In the second generation, all behaviours having their associated numbers in the *targeted_list* are translated to process definitions, in order to have the proper tree to be translated to any of the two forms mentioned above.

Note that this second tree generation is done again by using the inference rules and the internal representation of the specification as for the intermediate one.

However, the information gathered during the generation of this tree regarding the *target* nodes is used.

Note that it is necessary to use the inference rules again because the names of the parameters are different the second time. One could think that this could require a simple change of parameter names. However the mechanism of keeping track of these changes is complicated enough that we have found it easier to reapply the inference rules.

The corresponding *symbolic tree* of the example 31 shown in the previous section after the second generation:

Example 35 Buffer3 symbolic tree

```
1  bh0 * 1 in1?bit@1:bit [16]
2  bh1 * | 1 i (hiding: m1 !var_1) [17,16]
3  bh2 * | | 1 in1?bit@2:bit [16]
4  bh3 * | | | 1 i (hiding: m2 !var_2) [17,16]
5  bh4 * | | | | 1 i (hiding: m1 !var_1) [17,16]
6  bh5 * | | | | | 1 in1?bit@3:bit [16]
7  bh6 * | | | | | | 1 out !var_2 [17] ==> again bh3
8      * | | | | | 2 out !var_2 [17] ==> again bh2
9      * | | | | 2 out !var_2 [17] ==> again bh1
10     * | | 2 i (hiding: m2 !var_1) [17,16]
11 bh7 * | | | 1 in1?bit@2:bit [16] ==> again bh4
12     * | | | 2 out !var_1 [17] ==> again bh0
```

Symbolic Tree Internal Representation

The symbolic tree is generated simultaneously in the external form shown above and in an internal form, which already contains the information necessary for further transformation

Our *symbolic trees* are made up of *behaviours*, which correspond to *nodes* in the tree, and *actions*, which correspond to *edges* in the tree. The symbolic tree shown in the previous section is used to clarify the internal form representation.

Behaviours The behaviours in the *beh list* discussed in the previous section have the following form:

bh_save(Bc, I, Type),

where

Bc	is the cleaned behaviour.
I	is a unique number associated with the behaviour and is used as a reference in case of matching.
Type	is the behaviour type.

The type can be one of the following:

- dead, the behaviour is a deadlock.
- normal, otherwise; some normal behaviours are associated with a count "n" of the number of references to them "label(n)".

When a 'normal' behaviour becomes a *target* for the first time, its associated type is changed from 'normal' to 'label(1)'. This is incremented by 1 for every new *transfer*.

The following are the behaviours saved during the generation of the symbolic tree for example 31:

```
bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 instance(p5,[in1,m1],[]),[m1],seq([m2,s3],instance(p5,[m1,m2],
 [])),[m2],seq([out,s3],instance(p5,[m2,out],[]))),5,normal).


bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 seq([m1,s3],instance(p5,[in1,m1],[])),[m1],seq([m2,s3],
 instance(p5,[m1,m2],[])),[m2],seq([out,s3],instance(p5,[m2,out],
 [])),6,normal).


bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 seq([m1,s3],instance(p5,[in1,m1],[])),[m1],seq([m2,s3],
 instance(p5,[m1,m2],[])),[m2],instance(p5,[m2,out],[])),3,
 label(1)).


bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 instance(p5,[in1,m1],[]),[m1],seq([m2,s3],instance(p5,[m1,m2],
 [])),[m2],instance(p5,[m2,out],[])),2,label(1)).
```

```
bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 seq([m1,s3],instance(p5,[in1,m1],[])),[m1],instance(p5,[m1,m2],
 [])),[m2],instance(p5,[m2,out],[]))),1,label(1)).

bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 instance(p5,[in1,m1],[]),[m1],instance(p5,[m1,m2],[])),[m2],
 seq([out,s3],instance(p5,[m2,out],[])))),7,normal).

bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 seq([m1,s3],instance(p5,[in1,m1],[])),[m1],instance(p5,[m1,m2],
 [])),[m2],seq([out,s3],instance(p5,[m2,out],[])))),4,label(1)).

bh_save(hiding([m1,m2],parcomp(selected,parcomp(selected,
 instance(p5,[in1,m1],[]),[m1],instance(p5,[m1,m2],[])),[m2],
 instance(p5,[m2,out],[]))),0,label(1)).
```

These are the eight behaviour expressions labeled `bh0 ... bh7` in the symbolic tree of example 35.

The last `bh_save` is shown in different fonts, where the regular font corresponds to the cleaned behaviour (Bc), the bold font corresponds to the behaviour number (0 is the original behaviour's number, that is, this behaviour expression is `bh0` in example 35), and the italic font corresponds to the behaviour type. We shall not annoy the reader with the details of our internal notation, suffice is to say that, in external LOTOS notation, the behaviour is:

```
hide m1,m2 in
     ( (  buffer1 [ in1 ,m1 ]
        |[ m1 ]|
          buffer1 [ m1 , m2 ]
       )
        |[ m2 ]|
      buffer1 [ m2 , out ]
     )
```

Edges Edges correspond to the "actions" and the "again" in the symbolic tree. During the generation of the symbolic tree, the edges reached are saved in the following form:

edge(F, N, Action, Type)

where

F	is the father edge's number.
N	is a unique number associated with the edge.
Action	is the action in internal form representation, or is a reference to the target process.
Type	is the action's type.

There are two types of edges: edges referring to actions, and "again" edges. The edges referring to actions can be *internal, exit, dead, continue or normal*. The "again" edges refer to a *transfer*, and the argument "Action" associated with this edge is the reference to the *target* process.

The following are the edges saved during the generation of the symbolic tree (example 35):

```
edge(start,0,start,start).
edge(0,1,[g7,[[$,'bit@1',v14,s3]],[]],normal).
edge(1,2,i(hide,[g10,[[#,val(var_1,v332),s3]],[]]),normal).
edge(2,3,[g7,[[$,'bit@2',v14,s3]],[]],normal).
edge(3,4,i(hide,[g11,[[#,val(var_2,v332),s3]],[]]),normal).
edge(4,5,i(hide,[g10,[[#,val(var_1,v332),s3]],[]]),normal).
edge(5,6,[g7,[[$,'bit@3',v14,s3]],[]],normal).
edge(6,7,[g12,[[#,val(var_2,v332),s3]],[]],normal).
edge(7,8,3,again).
edge(5,9,[g12,[[#,val(var_2,v332),s3]],[]],normal).
edge(9,10,2,again).
edge(4,11,[g12,[[#,val(var_2,v332),s3]],[]],normal).
edge(11,12,1,again).
edge(2,13,i(hide,[g11,[[#,val(var_1,v332),s3]],[]]),normal).
edge(13,14,[g7,[[$,'bit@2',v14,s3]],[]],normal).
edge(14,15,4,again).
edge(13,16,[g12,[[#,val(val(var_1,v332),v14),s3]],[]],normal).
edge(16,17,0,again).
```

The reader should check that, after the "start" edge, the following seventeen edges correspond to the lines of the symbolic tree of example 35. Lines containing an "action" and an "again" are split into two. The last `edge` is shown in different fonts, where the regular font corresponds to the father edge's number (16), the bold font corresponds to the edge number, the italic font corresponds to the action, or is a reference to the *target* in the case of "again", and the bold-italic font corresponds to the action's type (in this case, the current edge is a reference to edge number `0`). This `edge` corresponds to `again bh0` in the symbolic tree. Again, we shall avoid the details of the internal representation, except for saying

that the previous edge (edge number 16) corresponds to the action `out !var_1` in the symbolic tree (example 35). Where `g12` represents gate `out`, `#` stands for `!` and the following expression stands for `var_1`.

Definitions and Instantiations *Transfer* or '`again`' nodes in the symbolic tree are represented by an `instance` in the internal form associated with the actual parameters, and *target* nodes are represented by a `definition` associated with the formal parameters.

Once a behaviour which has been found to be repeated is cleaned, all value identifiers, symbolic values and actual parameters appearing in the behaviour are saved in a list with their replacement identifiers in one of the following forms:

In the case of process definition (*target*):

definition(N, [[P_1, A_1, S_1], ... , [P_m, A_m, S_m]]),

where

- N is the *target* number' ,
- P_i is the new identifier replacing A_i (formal parameter),
- A_i is the value expression of the actual parameter 'i' in internal form representation (actual parameter) and
- S_i is the internal name of the sort of the actual parameter i.

The following are the definitions saved during the generation of the symbolic tree (example 35):

```
definition(0,[]).
definition(1,[[val(var_1,v332),val('bit@1',v14),s3]]).
definition(2,[[val(var_1,v332),val(val(var_1,v332),v14),s3]]).
definition(3,[[val(var_1,v332),val('bit@2',v14),s3],[val(var_2,
 v333),val(var_1,v332),s3]]).
definition(4,[[val(var_1,v332),val(var_1,v332),s3],[val(var_2,
 v333),val(val(var_2,v333),v14),s3]]).
```

Note that there is a definition for each one of the five behaviours to which there is an "again" in the symbolic tree in example 35, i.e. `bh0` to `bh4`.

We explain in detail the second of these definitions. There is only one parameter associated with this definition. The regular font, the target number, shows that it corresponds to `bh1`. The bold font gives the formal parameter of what will become subtree `tree<1>`. The italic font gives what will become the

actual parameter of the first instantiation of the same subtree. We shall see how this definition is used in the next section.

`Val(x,y)` is a Prolog predicate where `x` is the value expression of the actual parameter, and `y` is its internal name. In the example above (bold font), the internal name `v332` is provided, and the predicate returns the value of the parameter, which is `var_1`.

In the case of process instantiation (*transfer*):

instance(N_1, N_2, [[P_1, A_1, S_1], ... , [P_m, A_m, S_m]]),

where N_1 is the *target*'s number, N_2 is the edge father's number, and P_i, A_i, S_i are as above.

The following are the instantiations saved during the generation of the symbolic tree (example 35):

```
instance(1,11,[[val(var_1,v332),val(var_1,v332),s3]]).
instance(3,7,[[val(var_1,v332),val('bit@3',v14),s3],[val(var_2,
 v333),val(val(var_1,v332),v14),s3]]).
instance(2,9,[[val(var_1,v332),val(val(var_1,v332),v14),s3]]).
instance(4,14,[[val(var_1,v332),val('bit@2',v14),s3],[val(var_2,
 v333),val(val(var_1,v332),v14),s3]]).
instance(0,16,[]).
```

`instance` is used to define the actual parameters when there is a `tree` instantiation. Note that, since in our sample specification there is exactly one process instantiation for each process definition, we have exactly as many `definitions` as there are `instances`. However in general we could have more `instances` if there are several transfers to a given target.

The first `instance` is shown in different fonts. The regular font corresponds to the `tree`'s number (the target's number, in this case `bh1`). The bold font is the edge father's number (line 13 of the `edge` list above) and is used to distinguish between two `instances` referencing the same `target` behaviour. The bold-italic font is the internal form of the new value identifier (formal parameter). The italic font is the actual parameter which was replaced by the new value identifier (formal parameter) and the regular font is the sort of the parameter.

The external form of this tree is given in the following section.

Derivation of the parameterized tree

The symbolic tree is further transformed to parameterized tree.

The derivation process consists of transforming every node which represents a state, having a duplicate or a syntactically equivalent, (in other words, every target node), into an independent subtree having a definition of the form:

tree <n> (P_1:S_1, ... , P_m:S_m), where 'n' is the state number and P_i is the formal parameter <i> of type Si used in the subtree. These values can be recovered from the internal symbolic tree (definition(n, [[P_1, A_1, S_1], ... , [P_m, A_m, S_m]])).

Proceeded by a subtree instantiation of the form:

tree <n> (A_1, ... , A_m), where A_i is a value expression appearing in subtree <n> which was replaced by P_i in the second generation of the symbolic tree, these values can also be recovered from the symbolic tree (definition(n, [[P_1, A_1, S_1], ... , [P_m, A_m, S_m]])).

All nodes which represent a state which is either a duplicate or syntactic equivalent to state 'n', are transformed to a subtree instantiation tree <n>(A_1, ... , A_m), these values can be recovered from the symbolic tree (instance(n, SN, [[P_1, A_1, S_1], ... , [P_m, A_m, S_m]])).

Internal actions resulting from hiding and enable operators are represented in the following form:

i(<exp>), where <exp> is the operator name followed by the exited values, if any, in the case of enable, and the hidden action in the case of hide operator.

Example 36 Parameterized trees for our examples

This is the parameterized tree of the LOTOS specification given in example 31:

```
tree<0>
bh0 * in1 ?bit@1:bit [16] ==>tree<1>(bit@1)
tree<1>(var_1:bit)
bh1 * | i (hiding: m1 !var_1) [17,16] ==>tree<2>(var_1)
tree<2>(var_1:bit)
bh2 * | | in1 ?bit@2:bit [16] ==>tree<3>(bit@2,var_1)
tree<3>(var_1, var_2:bit)
bh3 * | | | i (hiding: m2 !var_2) [17,16] ==>tree<4>(var_1,var_2)
tree<4>(var_1, var_2:bit)
bh4 * | | | | i (hiding: m1 !var_1) [17,16]
bh5 * | | | | | in1 ?bit@3:bit [16]
bh6 * | | | | | | out !var_2 [17] ==>tree<3>(bit@3,var_1)
    * | | | | | out !var_2 [17] ==>tree<2>(var_1)
    * | | | | out !var_2 [17] ==>tree<1>(var_1)
    * | | i (hiding: m2 !var_1) [17,16]
bh7 * | | | in1 ?bit@2:bit [16] ==> tree<4>(bit@2,var_1)
    * | | | out !var_1 [17] ==>tree<0>
```

To explain how we arrived at this tree, we shall explain how one process definition (`tree<1>(var_1:bit)`), and one process instantiation (`tree<3>(bit@3,var_1)` at `bh6`) are obtained.

Note first of all that, as we transform the symbolic tree into parameterized tree, each process definition must be immediately preceded by an instantiation of the same process (in order to be able to get to it the first time). In addition, of course there must be other instantiations of the same process, which are represented by transfer nodes.

The parameterized tree is derived by following the saved `edges` top to bottom. For example, once we get to

`edge(`**`1`**`,2,i(hide,[g10,[[#,val(var_1,v332),s3]],[]]),normal)`

where the bold parameter refers to the father number **1**, which happens to be a *target* node. This indicates that the rest of the tree should be transformed into a new subtree of name `tree<`**`1`**`>`. At this point, we refer to the saved definitions, which, as mentioned above, contain both the formal parameters of the subtree `tree<1>`, and the actual parameters of the first instantiation of the same subtree. This definition is as follows:

```
definition(1,[[val(var_1,v332),val('bit@1',v14),s3]])
```

First of all, we compile the first instantiation of the subtree. The actual parameter `val('bit@1',v14)` is translated to its external form and added to the process instantiation `tree<1>(bit@1)` (at bh0), which connects the previous subtree `tree<0>` to the new one `tree<1>`.

Then we compile the definition for the subtree. This new subtree will have the following process definition :

```
tree<1>(var_1:bit)
```

Where the formal parameter `(var_1:bit)` is the bold font (P_i) in the saved `definition` shown above.

Finally, we give an example of how an "again" edge is transformed into process instantiation. Once we get to the "again" edge

```
edge(7,8,3,again)
```

we refer to the saved `instances`

```
instance(3,7,[[val(var_1,v332),val('bit@3',v14),s3],
 [val(var_2,v332),val(val(var_1,v332),v14),s3]])
```

The actual parameters A_i (bold-italic font) are translated to their external representation and added in the same order to the process instantiation `tree<3>`.

The resulting instantiation is:

```
tree<3>(bit@3,var_1)
```

The parameterized tree of the LOTOS specification given in example 33

```
tree<0>
bh0 * g ?nat@1:nat [11]
bh1 * | i [11] ==> tree<2>(succ(nat@1),nat@1)
tree<2>(var_1, var_2:nat)
bh2 * | | [var_1>var_2] g ?nat@2:nat [15] ==> tree<2>(nat@2,succ(var_2))
    * | | [var_1<var_2] g ?nat@2:nat [17] ==> tree<2>(var_1,nat@2)
    * | | [var_1=var_2] g !0 [19]        DEADLOCK
```

The parameterized tree of the LOTOS specification given in example 34

```
tree<0>(0)
tree<0>(var_1)
bh0 * g !var_1
        [succ(0)=var_1] [8,10]
bh1 * | i [8,10]  ==> tree<0>(succ(var_1))
    * g1 [13]
bh2 * | i [13]  ==> tree<0>(succ(var_1))
```

Derivation of LOTOS Monolithic Specification

The derivation of LOTOS monolithic specifications is done in the same way as the derivation of the parameterized tree with some slight differences.

In the derivation process to monolithic style specification, five different types of tree nodes have to be considered:

1. Deadlock nodes which indicate the end of a path are replaced by a *stop*.
2. All nodes which represent states having a duplicate or a syntactic equivalent are transformed to a process definition-instantiation pair, namely:
   A process definition of the form:
   `process P<n>` (P_1:S_1, ... , P_m:S_m), where 'n' is the state number and P_i is the formal parameter <i> of type S_i used in the process. These values can be recovered from the internal symbolic tree (definition(n, [[P_1, A_1, S_1], ... , [P_m, A_m, S_m]])).
   Preceded by a process instantiation of the form:
   `P<n>` (A_1, ... , A_m), where A_i is a value expression appearing in process <n> which was replaced by P_i in the second generation of the

symbolic tree. These values can also be recovered from the symbolic tree (instance(n, SN, [[P_1, A_1, S_1], ... , [P_m, A_m, S_m]])).

3. All nodes, which represent a state which is a duplicate or a syntactic equivalent of state n, are transformed into process instantiations,
4. Internal actions are represented as 'i'.
5. External and continue nodes are represented in the same form as they appear in the symbolic tree.

Example 37 Specification buffer3_mono

The corresponding monolithic style specification of the above LOTOS specification:

```
 1  specification buffer3_mono [in1,out] : noexit
 2
 3    behaviour
 4         P0[in1,out]
 5    where
 6  process  P0[in1,out]:noexit :=
 7
 8   in1 ?bit@1:bit ;
 9      P1[in1,out](bit@1)
10
11  endproc
12
13  process  P3[in1,out](var_1, var_2:bit):noexit :=
14
15   i (hiding: m2 !var_2) ;
16      P4[in1,out](var_1,var_2)
17
18  endproc
19
20  process  P2[in1,out](var_1:bit):noexit :=
21
22   in1 ?bit@2:bit ;
23      P3[in1,out](bit@2,var_1)
24   []
25
```

Example 37 (Continued) Specification buffer3_mono

```
26   i (hiding: m2 !var_1) ;
27      (
28      in1 ?bit@2:bit ;
29         P4[in1,out](bit@2,var_1)
30      []
31
32      out !var_1 ;
33         P0[in1,out]
34      )
35
36  endproc
37
38  process  P1[in1,out](var_1:bit):noexit :=
39
40   i (hiding: m1 !var_1) ;
41      P2[in1,out](var_1)
42
43  endproc
44
45  process  P4[in1,out](var_1, var_2:bit):noexit :=
46
47   i (hiding: m1 !var_1) ;
48      (
49      in1 ?bit@3:bit ;
50         out !var_2 ;
51            P3[in1,out](bit@3,var_1)
52      []
53
54      out !var_2 ;
55         P2[in1,out](var_1)
56      )
57   []
58
59   out !var_2 ;
60      P1[in1,out](var_1)
61
62  endproc
```

Example 37 (Continued) Specification buffer3_mono

```
63
64  TYPE bit1 IS
65     SORTS bit
66     OPNS 0,1  : -> bit
67      inc : bit -> bit
68     EQNS
69     OFSORT bit
70     inc(0) = 1 ;
71     inc(1) = 0 ;
72  ENDTYPE
73 ENDSPEC
```

One can check that this specification corresponds to the diagram in figure 8.

The corresponding monolithic style specification of the example 33:

```
 1  specification Nat_numbers_test[g] : noexit
 2    behaviour
 3         P0[g]
 4    where
 5  process  P0[g]:noexit :=
 6   g ?nat@1:nat ;
 7      i ;
 8         P2[g](succ(nat@1),nat@1)
 9  endproc
10  process  P2[g](var_1, var_2:nat):noexit :=
11   [var_1>var_2] g ?nat@2:nat ;
12      P2[g](nat@2,succ(var_2))
13   []
14   [var_1<var_2] g ?nat@2:nat ;
15      P2[g](var_1,nat@2)
16   []
17   [var_1=var_2] g !0 ;
18      stop
19  endproc
20
21  library Boolean Natural endlib
22
23  endspec
```

The corresponding monolithic style specification of the example 34:

```
 1  specification Nat_numbers_test[g,g1] : noexit
 2    behaviour
 3         P0[g,g1](0)
 4    where
 5  process  P0[g,g1](var_1:nat):noexit :=
 6   g !var_1
 7     [succ(0)=var_1]  ;
 8     i ;
 9        P0[g,g1](succ(var_1))
10   []
11   g1 ;
12     i ;
13      P0[g,g1](succ(var_1))
14  endproc
15
16  type Nat_numbers is
17
18       sorts nat
19
20       opns 0:         -> nat
21            succ: nat -> nat
22
23  endtype
24  endspec
```

Section 5 Symbolic Tree Reduction

Congruence rules (chap. 2) can be used to reduce behaviour trees by removing some internal events and even some branches. Currently, only the following rules are used in our system:

$$1 - \ a;i;B \overset{c}{\approx} a;B$$
$$2 - \ B\,[\,]i;B \overset{c}{\approx} i;B$$
$$3 - \ a;\,(B[\,]B) \overset{c}{\approx} a;B$$

where a denotes any action.

It is possible, however, to extend the system to include additional rules.

The following algorithm applies the congruence rules on the monolithic style specification in a bottom-up order. In other words, these rules are applied on a node once all its subnodes are reduced. The algorithm starts on leaf nodes, i.e. “stop” and “again” nodes.

Once a rule is applied, the tree is updated in order to maintain the corresponding connections between nodes.

Algorithm 4.1

```
Reduce-tree(node)
   Forall the branches of this node starting
   with the left most do
      If (branch is normal or internal) then
         Reduce_tree(branch)
         Apply_rules(sub_node)
      EndIf
   End Forall
End Reduce_tree
Apply_rules(node)
   If (# of branches < 2) then
      Attempt_rule_1
   Else
      On every combination of pairs of branches do
         Attempt_rule_2
         Attempt_rule_3
   EndIf
End Apply_rules
```

Where `Attempt_rule_`*I* (where *I* = 1,2,3) consists in checking whether rule *I* applies and, if so, applying it.

Example 38 Symbolic Tree Reduction

This is the monolithic style specification of the LOTOS specification given in example 31, after the application of the algorithm on the symbolic tree. The resulting tree can be also transformed into a parameterized tree. Figure 9 shows the corresponding diagram.

```
1  specification buffer3_mono[in1,out] : noexit
2
3  behaviour
4       P0[in1,out]
5  where
6  process  P0[in1,out]:noexit :=
```

Example 38 (Continued) Symbolic Tree Reduction

```
 7
 8   in1 ?bit@1:bit ;
 9      P1[in1,out](bit@1)
10
11  endproc
12
13  process  P3[in1,out](var_1, var_2:bit):noexit :=
14
15   i (hiding: m1 !var_1) ;
16      (
17      in1 ?bit@3:bit ;
18         out !var_2 ;
19            P3[in1,out](bit@3,var_1)
20      []
21
22      out !var_2 ;
23         P1[in1,out](var_1)
24      )
25   []
26
27   out !var_2 ;
28      P1[in1,out](var_1)
29
30  endproc
31
32  process  P1[in1,out](var_1:bit):noexit :=
33
34   in1 ?bit@2:bit ;
35      P3[in1,out](bit@2,var_1)
36   []
37
38   i (hiding: m2 !var_1) ;
39      (
40      in1 ?bit@2:bit ;
41         P3[in1,out](bit@2,var_1)
42      []
43
```

Example 38 (Continued) Symbolic Tree Reduction

```
44      out !var_1 ;
45         P0[in1,out]
46      )
47
48  endproc
49
50  TYPE bit1 IS
51     SORTS bit
52     OPNS 0,1  : -> bit
53      inc : bit -> bit
54     EQNS
55     OFSORT bit
56     inc(0) = 1 ;
57     inc(1) = 0 ;
58  ENDTYPE
59 ENDSPEC
```

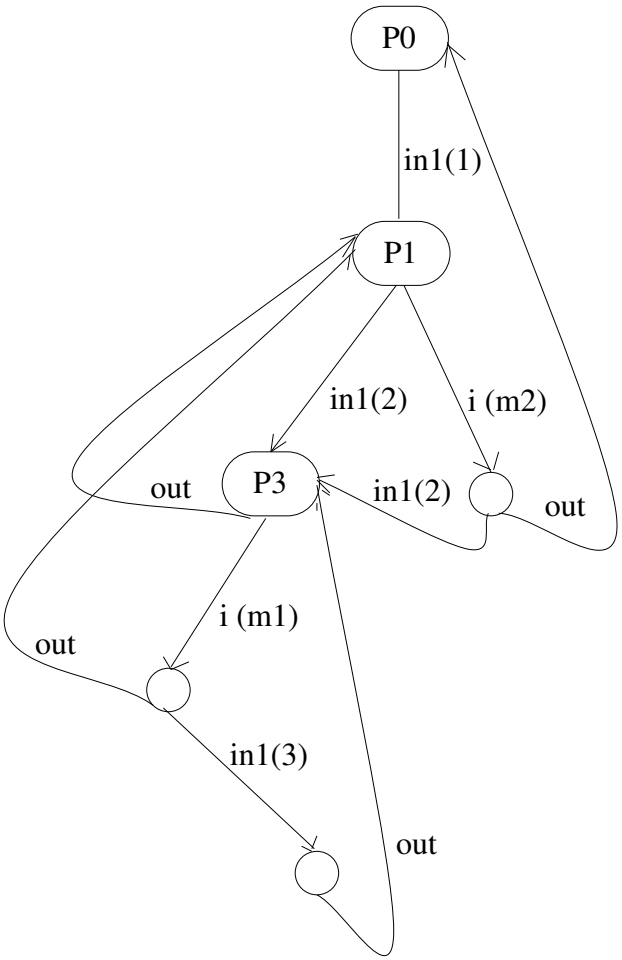


Figure 9 Diagram of the monolithic style specification (ex. 31)

By comparing both monolithic specifications, the reduced and the normal, we see that some internal actions have been deleted without changing the external behaviour described by the specification. As a result, some processes could also be eliminated, and their instantiations could be replaced by their bodies.

Note that our transformation could be considered as an automated "proof" of the fact that specification `buffer3` actually specifies a 3–place buffer, because the monolithic specification clearly enumerates all possible behaviours of the buffer.

This is the reduced monolithic style specification of example 33:

```
specification Nat_numbers_test[g] : noexit
  behaviour
       P0[g]
  where
process  P0[g]:noexit :=
 g ?nat@1:nat ;
       P2[g](succ(nat@1),nat@1)
endproc
process  P2[g](var_1, var_2:nat):noexit :=
 [var_1>var_2] g ?nat@2:nat ;
    P2[g](nat@2,succ(var_2))
 []
 [var_1<var_2] g ?nat@2:nat ;
    P2[g](var_1,nat@2)
 []
 [var_1=var_2] g !0 ;
    stop

endproc

library Boolean Natural endlib

endspec
```

This is the reduced monolithic style specification of example 34:

```
specification Nat_numbers_test[g,g1] : noexit
  behaviour
       P0[g,g1](0)
  where
process  P0[g,g1](var_1:nat):noexit :=
 g !var_1
   [succ(0)=var_1]  ;
     P0[g,g1](succ(var_1))
 []
 g1 ;
    P0[g,g1](succ(var_1))
endproc

type Nat_numbers is

     sorts nat

```

```
18       opns 0:         -> nat
19            succ: nat -> nat
20
21  endtype
22  endspec
```

Chapter 5 Application of our Work

This chapter is devoted to the description of the possible applications of our work. The first section defines conformance testing. Then, the concept of deriving test sequences from LOTOS trees is explained. This is followed by a short discussion of TTCN. Section 4 further explains the process of transforming the LOTOS trees into TTCN. Finally, an example is given on the generation of test cases and their translation to TTCN.

Section 1 Conformance Testing

Conformance testing is an area of protocol development methodology that deals with the problem of testing if an implementation of a standard *conforms* to the standard specification. Conformance tests have to cover all aspects stated in the standard of a protocol. Most test methods for protocols derive test suites manually from informal or semi-formal descriptions. The use of formal description techniques enables us to develop automatic or semi-automatic methods and test tools for the purpose of generating *conformance tests* from formal specifications. An ESTELLE-based test method is described in [Ura87]. In what follows, we shall discuss the relevance of our tool in the generation of *conformance tests* from LOTOS specifications.

The subject of this chapter is an ongoing topic in the area of formal methods in conformance testing [CCITT/ISO]. In particular, there is a substantial literature on test derivation from LOTOS specifications [Gue89]. Unfortunately, it is not possible for us to give a complete treatment of this complex topic. We will limit ourselves to provide some general ideas on the subject, and the development of these ideas must be left to further research.

Section 2 LOTOS Trees and Test Sequence Generation

The symbolic tree generated by SELA, resulting from the expansion of a given LOTOS specification, is a tree of alternatives. This tree explicitly indicates all possible execution sequences of the entity specified, up to a user-defined maximum length. It can be used as a test tree by reversing the direction of the data flow in the symbolic tree, due to the assumption that the environment of the

specification is the tester. The tester behaves the opposite way as the symbolic tree, a receive (symbol ?) in the symbolic tree corresponds to a send (symbol !) for the tester ; and a send (symbol !) in the symbolic tree corresponds to a receive (symbol ?) for the tester.

This technique appears to be valuable for conformance testing. It makes it possible to extract test cases directly from formal descriptions, eliminating or reducing the importance of the interpretation of the informally specified standard. The automatic or semi-automatic generation of interactions from a formal specification appears to be possible by using the information contained in the selection predicates [GL89].

Section 3 Discussion of TTCN

TTCN (Tree and Tabular Combined Notation) is a framework and methodology as well as a semi-formal notation developed within ISO to specify conformance test suites for various protocols [Int89]. There is an undergoing activity to standardize these test suites to be used by test centers.

TTCN also serves the following purposes:

— It provides a common reference for assessing other notations and assists in examining the problems arising in testing and test suite design;
— It provides a basis for the translation of tests into other notations;
— It can be used to develop trial test suites.

TTCN is provided in two forms:

— A graphical (TTCN-GR) form, more suitable for human readability;
— A machine processable form (TTCN-MP), suitable for transmission of TTCN descriptions between machines and possibly suitable for other automated processing.

The difference between these two forms lays in the syntax. TTCN-MP uses keywords instead of boxes for information delimiters.

A test suite can be seen as a hierarchy ranging from the complete test suite down to test events. A test suite contains test groups which contain test subgroups, containing test cases, which consist of test steps which include test events.

A test suite written in TTCN has four parts:

— Test suite overview part,
— Declaration part,
— Dynamic part and
— Constraint part.

Those parts are described hereafter.

Test suite overview

This section shall include at least the following information:

— Full references to the protocol or protocols whose implementations are the subject of the test suite.
— A precise indication of the test method to which the test suite applies.
— An overview and list of contents of the test suite by test subgroups and test group.
— A complete index to the test cases, test steps and other parts of the test suite.

Declaration part

The purpose of the declaration part is to describe the set of events, and other attributes to be used in the test suite. These include Abstract Service Primitives (ASPs) which occur at Points of Control and Observation (PCOs), timer identifiers, user defined types and operators, test suite parameters, global variables and constants, PCOs, Protocol Data Units (PDUs) and their parameters and any abbreviations used in the test suite specification. Formats are available for each of these items.

Dynamic part

The dynamic part contains the main body of the test suite and the test case or test step behaviour descriptions. It consists of dynamic behaviour tables, which contain the following information for the behaviour description of each test case, test step or set of related test steps:

□ *A reference*: the reference gives a name to the test case or step behaviour description, and it shall reflect the test case position in the hierarchy of the test suite.

- *An identifier*: the identifier provides a unique and concise reference to a test case, test step, or set of related test steps.
- *A statement of purpose*: This shall be an informal statement of purpose of the test case or test step. For test steps, references to possible preceding or following test steps shall be given.

The dynamic behaviour tables contain also columns for behaviour descriptions, labels, constraints to particular ASPs or PDUs, verdicts and comments.

In the behaviour tree, a set of alternatives is expressed by writing the first symbol of each alternative aligned in the same column (level of indentation). Test events initiated at named PCOs (sent events) are identified by the symbol '!', and test events accepted (received events) at named PCOs are identified by the symbol '?'.

Synchronization constraints can be used to synchronize events at distinct PCOs. Other test events used in the behaviour tree are pseudo-events, such as OTHERWISE , TIMEOUT and ELAPSE. OTHERWISE denotes the reception of any event which is not explicitly listed among the alternatives preceding the OTHERWISE. TIMEOUT is used to check the expiration of the specified Timer. ELAPSE provides a means of checking how long the execution of a test case remains cycling around a set of alternatives without any of them matching and exiting from that set of alternatives if the prescribed time (TD) has elapsed.

Test events may be associated with expressions. There are two kinds of expressions: *assignment* expressions and *boolean* expressions. An event is qualified by a boolean expression when that expression is written between square brackets. An event contains an assignment expression when that expression is written between parentheses.

A set of operations, such as START and CANCEL, are used to model the timer management. Trees may be attached to other trees by inserting the name of the tree to be attached, prefixed by the symbol '+'. A label may be placed in the label column for an event. A jump to a label 'A' is specified by the notation ' -> A' or 'GOTO A'. The repeat statement is a mechanism allowing the specifier to iterate a test step, a variable number of times.

The constraint reference column enables reference to be made to a specific constraint placed on an ASP or a PDU. Such constraints are defined in the constraint part.

Entries in the verdict column indicate a verdict of the test case. Three kinds of verdicts are available:

- *Pass*: some aspect of the test purpose has been achieved.
- *Inconclusive*: something has occurred which makes the test case inconclusive for some aspect of the test case.
- *Fail*: a protocol error has occurred or some aspect of the test purpose has resulted in failure.

Constraint Part

This part allows the description of the value coding of parameters in ASPs and PDUs. TTCN provides formats for such specifications. Specific binary or hexadecimal values are assigned to ASPs and PDUs parameters. In some cases, it is not required to specify a definite value for a field. In this case, one can represent that value by a free value which is considered to be global to the test suite, and shall be declared at the beginning of the constraints part. Reference to particular encodings is made in the constraints reference column of the tables used in the Dynamic Part.

Section 4 Translation of LOTOS Trees to TTCN

There is no automated procedure to translate a LOTOS tree to TTCN. This is an undergoing research topic, which could not be discussed in detail in this thesis. However it was shown in [Gue89] how this can be done manually, by using as example a full specification of the X.25 LAP-B.

In principle, this can be done by translating the test sequences generated from LOTOS trees to a TTCN test suite, or by translating directly a LOTOS tree to a TTCN tree and then generating the test sequence of the translated TTCN tree (see figure 10). However these two methods should result in having the same test suites represented in TTCN form.

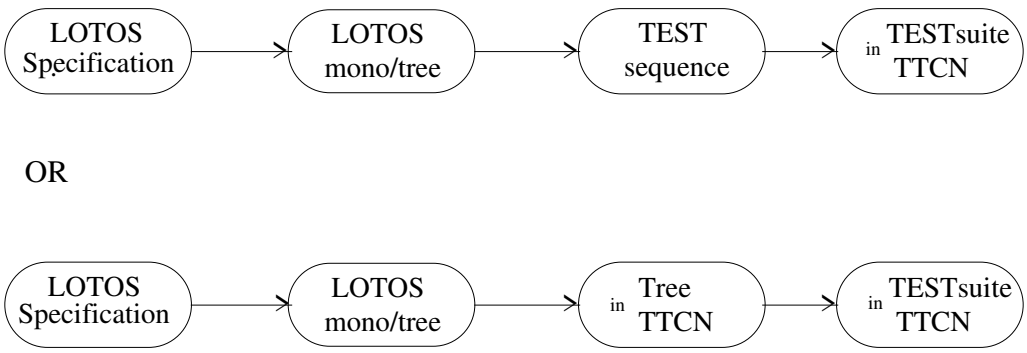


Figure 10 Test Sequence Derivation

In what follows, the first method has been adopted, where the test sequences are generated from the LOTOS symbolic trees and translated to TTCN.

The method we illustrate in this chapter applies in the simplified case of a LOTOS specification that is completely *deterministic*. *Nondeterminism* introduces some complexities in the testing problem, which are outside of the scope of this thesis. For discussion of more general test case generation methods that take nondeterminism into consideration, see [BSS87]. Therefore, the specifications considered in this chapter will contain no internal actions.

The test model we have adopted is a *direct drive*, where the tester synchronizes with the specification, acting as the environment, in order to drive the specification along the desired path. Also, nonexecutable test sequences can be generated due to the presence of symbolic values in the symbolic tree. As mentioned in section 4.3, the guards and selection predicates involving symbolic values were not evaluated when the symbolic tree was created. For instance, a branch guarded by a condition such as *succ(Nat@1)* < *Nat@1)* will have been preserved in the symbolic tree, eventhough this condition is *false* for any given value of *Nat@1*. Test cases derived from such branches are not executable.

Definitions

The global type declarations of a LOTOS specification contains all data types and Abstract Service Primitive (ASP) declarations in the declaration part of TTCN, since these declarations apply to all the tests in the test suite. Each

LOTOS type must be translated to an equivalent TTCN definition table, where the type identifier corresponds to the type name in TTCN.

The gates of the specification can be represented as Points of Control and Observation (PCO) in the declaration section of TTCN. The parameters of a LOTOS specification are translated to test suite parameters, since they globally parameterize the test suite. These translations are not always straightforward [MF91].

Behaviour Expression

The dynamic behaviour part of TTCN defines the test cases as behaviour trees using a tree structures syntax, thus the test cases extracted from the monolithic style specification can be translated to dynamic behaviour tables.

Due to the similarity in the syntaxes of TTCN dynamic behaviour tables and the monolithic style, the translation is straightforward. For example:

```
     LOTOS Construct                  TTCN Translation
     ---------------                  ----------------
Events:
      X ! ack                           ! ack
      X ? Bitstring@1:Bitstring         ? Bitstring
Sequence:
      X ! ack ;                         ! ack
        X ? Bitstring@1:Bitstring         ? Bitstring
Choice:
      L ! DISreq                        ! DISreq
   [] L ? ind:DISind                    ? DISind
Boolean Expression:
      X ? nat1:nat [nat1 > 2]          ? nat [nat > 2]
```

A test case extracted from a monolithic style specification can be defined as a number of subtrees in TTCN, which can be attached one to the other by substituting the name of the subtree to be attached, prefixed by **+**, for an event name. All final events of a test case have a verdict assigned to them in TTCN. The verdict can be one of the following: Pass, Fail or Inconclusive.

Section 5 Test Sequence Generation

The monolithic specification represents the valid behaviour of the IUT under test. Test cases are derived from the monolithic specification by processing each state[4] using a three step procedure:

1. **initialization**: creating a prefix sequence of actions that leads to the targeted state
2. **evaluation**: testing the transitions from the targeted state into the destination states
3. **termination**: driving the IUT after the transition performed in the evaluation step to the initial state

Note that we do not include state identification sequences, because we are not assuming that they exist.

Test cases consist of a preamble followed by a test body and ending with a postamble. The preamble consists of the initialization step. The test body consists of the evaluation step. The postamble consists of the termination step. The following example illustrates an abstract test case in TTCN:

```
+preamble
    +test_body_part 1
        +postamble 1
    +test_body_part 2
        +postamble 2
    .
    .
    .
    +test_body_part n
        +postamble n
```

Figure 11 Example of a test case structure in TTCN (Continued . . .)

[4] A LOTOS specification is a labelled transition system, with an initial state being the specification's main behaviour, and with all other states (possibly an infinity) being derived from the initial state on particular labels (i.e. events).

Figure 11 Example of a test case structure in TTCN

Initialization Step

The initialization step consists of finding an initialization path which drives the specification to a given state. Consider the specification `buffer3_mono` given in example 38. According to what we said above, we have simplified it by removing nondeterminism[5]. Since this specification consists of three processes, it can be represented by the following state diagram (Figure 12) using three different states `P0,P1 and P3`.

[5] Note that in this specific example, this does not change the test sequences obtained, because in general (a;B [] i;(a;B [] b;C)) is testing equivalent to (a;B [] b;C).

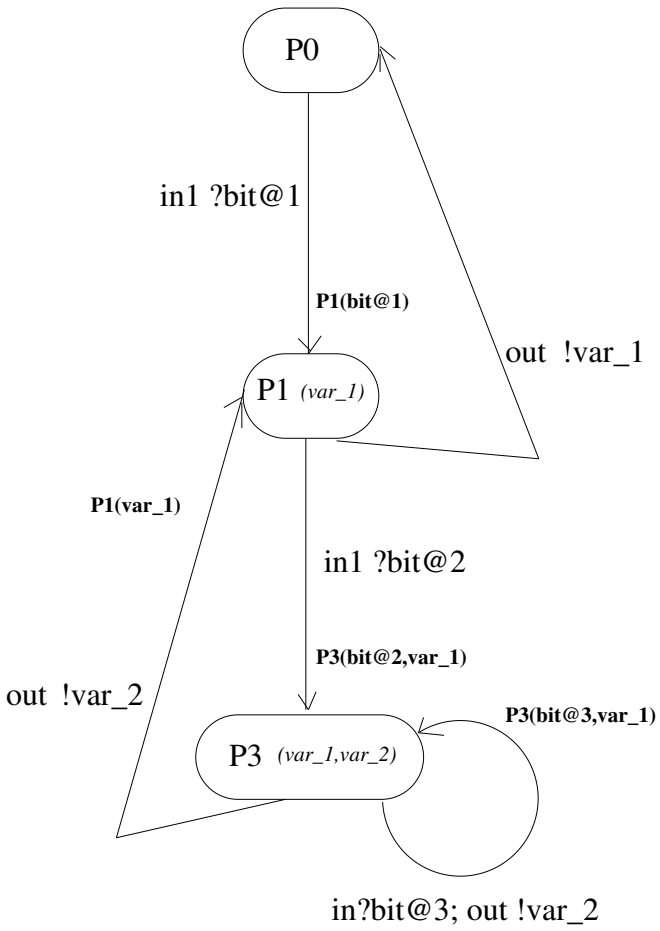


Figure 12 Diagram of specification *buffer3_mono*

where the normal font represents the actions, the bold font represents the instantiation of the next state (or process) associated with the actual parameters, and the italic font represents the formal parameters.

The initialization path (`ip0`) which drives the specification to state `P0` is empty since `P0` is the initial state in the specification. The initialization path (`ip1`) of state `P1` is obtained by following the edge which goes to state `P1`. Therefore the path (`ip1`) is:

```
ip1
            in1 ! bit@1
```

The initialization path (`ip3`) of state `P3` which drives the specification to state `P3` is obtained by concatenating path (`ip1`) with the edge that goes from state `P1` to state `P3`. Therefore the path `ip3` is:

```
ip3
       +ip1
          in1 ! bit@2
```

In general, the initialization paths of the next possible states are obtained by concatenating the initialization path of the current state with those edges which lead to them.

Evaluation Step

The evaluation step consists of performing all the transitions of the current state. As mentioned above in the test sequence generation (section 2), for a reception of a frame *fs* in the LOTOS specification, there is a transmission of the frame *fs* in the tester. Correspondingly, for a transmission of a frame *fr* in the LOTOS specification, there is a reception of the frame *fr* in the tester. The transmission of *fs* and the reception of *fr*, if any, represent the body of the test case.

The following is the test case extracted from state `P0` :

```
(1) in1 ! bit@1
       + tr1
```

Where `tr1` is the termination path of state `P1`, since this action would drive the specification to state `P1`.

The following are the test cases extracted from state `P1(var_1)`:

```
(1) + ip1
         out ? var_1
            + tr0
(2) + ip1
         in1 ! bit@2
            + tr3
```

Where tr0 and tr3 are the termination paths of states P0 and P3 respectively.

Similarly, the following are the test cases extracted from state P3(var_1,var_2):

```
(1) + ip3
          out ? var_2
             + tr1
(2) + ip3
          in1 ! bit@3
             out ? var_2
                + tr3
```

Termination Step

The termination step consists of finding a path to drive the specification from the current to the initial state (P0). Assuming that, this path will only lead to state (P0).

The termination path tr0 of state P0 is empty, since P0 is the initial state.

The following is a termination path of state P1(var_1):

```
tr1
(1)  out ? var_1
```

A termination path of state P3(var_1,var_2) is :

```
tr3
(1)  out ? var_2
       + tr1
```

The complete test sequence

By replacing the formal parameters by their actual values, we can construct the complete test cases for every transition. A complete test case of a transition emanating from state n is the initial path ipn followed by the test body for the transition emanating from state n, and the postamble that starts from the end state of the transition.

The complete test case for state P0 is:

```
(1)  in1 ! bit@1
        out ? bit@1
```

The complete test cases for state `P1(var_1)` are:

```
(2)   in1 ! bit@1
         out ? bit@1
(3)   in1 ! bit@1
         in1 ! bit@2
            out ? bit@1
                out ? bit@2
```

The complete test cases for state `P3(var_1,var_2)` are:

```
(4)   in1 ! bit@1
         in1 ! bit@2
            out ? bit@1
                out ? bit@2
(5) in1 ! bit@1
       in1 ! bit@2
          in1 ! bit@3
             out ? bit@1
                out ? bit@2
                   out ? bit@3
```

The second and fourth test cases would be deleted since they are similar to the first and third test cases respectively. As a result of the deletion we would have the following test cases :

```
(1)   in1 ! bit@1
         out ? bit@1
(2)   in1 ! bit@1
         in1 ! bit@2
            out ? bit@1
                out ? bit@2
(3) in1 ! bit@1
       in1 ! bit@2
          in1 ! bit@3
             out ? bit@1
                out ? bit@2
```

```
out ? bit@3
```

This is equivalent to testing that the buffer can input one, two, or three bits, and output them in the same order.

Test cases in TTCN

The translation to TTCN, to obtain the tester, proceeds as follows: if a frame *fr* is received by the tester after sending *fs*, the tester of what follows the reception of *fr* is added to the body of the test case after reception of *fr*. If no reception follows the sending of *fs*, the tester of the state where we arrived by the sending of *fs* is added to the body of the test case. If no reception follows the sending of *fs* and the latter has led back to the same state, the test of other alternatives is added to the test case after the timer starts and stops (to make sure nothing is received).

If a frame is to be received without sending an *fs*, and there is a second reception immediately afterwards, the tester of what follows is added to the body of the test case. If no reception follows the receive of *fr*, the tester of the state where we arrived by the receive of *fr* is added to the body of the test case. If no reception follows the receive of *fr* and the latter has led back to the same state, the test of other alternatives is added to the test case after the timer starts and stops (to make sure nothing is received).

Note that the 'no reception' case above covers three possibilities, that the next action is a sending action, or that there is no next action, because we have reached the end of the test sequence or a stop.

The timer Elapse TD is reset after every action, and if the timer stops before the next action is executed the test case would fail.

An otherwise is added and a verdict *fail* is issued at any point where something different from what is expected to be received can occur. All this is summarized in the following scheme:

```
X ! fs
   if reception
      X ? fr
         + tester of what follows
      X ? otherwise                                  fail
   else (no reception)
      if fs led  to next state
```

```
            + tester of next state
         else (fs led back to the same state)
            start Timer
               time-out Timer
                  + test other alternatives
               X ? otherwise                          fail
   X ? fr
      if reception
         + tester of what follows
      else (no reception)
         if fr led to next state
            + tester of next state
         else (fr led back to the same state)
            start Timer
               time-out Timer
                  + test other alternatives
               X ? otherwise                          fail
   X ? otherwise                                      fail
   ? Elapse TD                                        fail
```

The following are the test cases extracted from the above example:

```
(1)   in1 ! bit@1
         out ? bit@1                                 pass
         out ? Otherwise                             fail
         ? Elapse TD                                 fail
(2)   in1 ! bit@1
         in1 ! bit@2
            out ? bit@1
                out ? bit@2                          pass
                out ? Otherwise                      fail
            out ? Otherwise                          fail
            ? Elapse TD                              fail
(3) in1 ! bit@1
       in1 ! bit@2
          in1 ! bit@3
             out ? bit@1
                out ? bit@2
```

```
      out ? bit@3                    pass
      out ? Otherwise                fail
   out ? Otherwise                   fail
out ? Otherwise                      fail
? Elapse TD                          fail
```

Chapter 6 Conclusions and Suggestions for Future Work

Section 1 Summary of the Thesis

In this thesis, we have presented a method for the generation of the behaviour tree of a LOTOS specification (or a process). In order to do this, the specification is executed symbolically, that is, without the use of actual values. We have also explained how to identify repeated or syntactically equivalent behaviours. Detecting all (semantically) equivalent behaviours is not decidable [Mil89], however some heuristics to identify some cases of equivalences are shown.

We have shown how the symbolic tree can be reduced by application of congruence rules in order to reduce its size, without changing the semantics of the specification. This is done by identifying some obvious reductions, since in general this is also an undecidable problem [Mil89].

We have also discussed the translation of the symbolic tree into a parameterized tree, where the first occurrences of repeated or syntactically equivalent nodes are transformed to an independent subtree having a unique name and a list of parameters in order to make the tree easier to follow. Furthermore, we have presented a method for translating the symbolic tree into a monolithic style specification. This is done by transforming the first occurrence of a repeated or equivalent behaviour into a LOTOS process definition and the later occurrences into process instantiations.

All these methods were implemented in Prolog, and integrated to the University of Ottawa LOTOS toolkit.

We have demonstrated an informal method to generate test cases from the obtained monolithic specification. Finally, we have introduced TTCN, and discussed the possibility of translating the behaviour trees to TTCN.

Section 2 Limitations

In Section 3.3, we presented an algorithm for the generation of the symbolic tree and the heuristics used to detect the equivalences between behaviours. These heuristics are based on comparing behaviours. In general, this presupposes the use

of verification methods in order to prove equivalence of two behaviour expressions according to one of the various equivalence relations that have been studied for LOTOS. This is obviously beyond the scope of this thesis.

Other weaknesses of this method are:

1. There are only three congruence rules used in our system, and they are applied once the behaviour tree is generated.
2. The heuristics we have used are not able to deal with divergence in the specification (Figure 39).

Example 39 Divergence

```
P[a,b] := ( P[a,b]
            |||
             a ; stop
          )
```

In the execution of this specification, there is an infinite number of actions 'a' which is possible at the very beginning. An interpreter cannot deal with such a situation.

Section 3 Future Work

The congruence rules mentioned in the previous section could be applied while generating the symbolic tree, which can reduce the execution time. Furthermore, additional congruence rules could be applied to make the system more efficient in detecting equivalence between behaviours.

A possible next step would be to automate the process of generation of test cases based on the behaviour tree. This process has been informally discussed in chapter 5.

Another more challenging task would be to automate value generation to replace symbolic values with actual ones. In the example of chapter 5, this would mean generating actual values of sort bit in order to complete the test cases.

This has several applications:

- It would help in the generation of the test cases where values are involved.
- Contradictions in the behaviour tree would be detected, which is useful in order to validate the actual behaviour of a specification with respect to its intended behaviour.

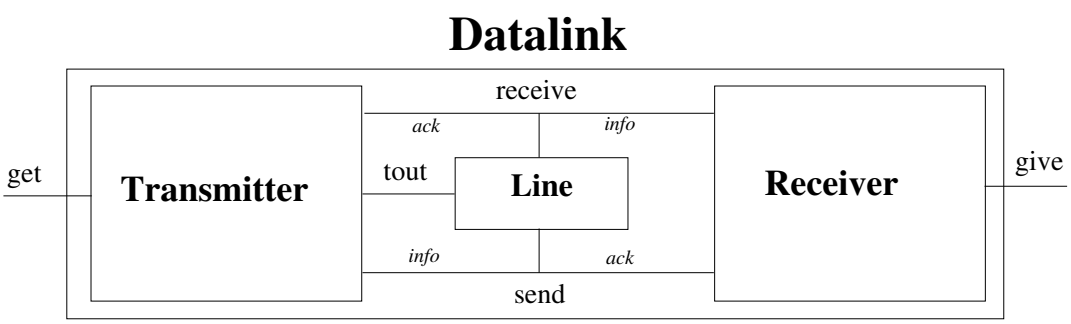
Finally, techniques to prune as much as possible the trees and to represent them compactly in internal representation will have to be studied, if specifications of real-life systems are to be handled.

Appendix A Case study : Datalink Service Provider

A Datalink Service Provider [QFP88] is presented below and includes one direction in the information flow. Only a **transmitter** entity connected through a semiduplex line to a **receiver** entity is presented. The error detection mechanism is the one of the familiar *alternating bit protocol*, i.e. possible values for the sequence number are 0 or 1.

The process **transmitter** sends the data, associated with a sequence number, after receiving it through the gate *get*, to process **Line**. A timer is set while the process **transmitter** is waiting for an acknowledgment. An acknowledgment with a wrong sequence number is ignored, and a timeout will follow. If the timer has timed out before receiving the acknowledgment, the same data is retransmitted, otherwise **transmitter** is ready to get another data item to be transmitted.

The process **Receiver** is ready to receive a data item from process **Line**, together with its sequence number. If the sequence number received is the expected number, the data is transmitted through the gate *give*, and an acknowledgment is sent to process **Line** associated with the sequence number of the next set of data. Otherwise, if the data received is old data, an acknowledgment is sent to process **Line**, associated with the expected sequence number.



Note that process **Line** synchronizes on gate "send" with process **Transmitter** when the message is an *info*, and with process **Receiver** when the message is an *ack*. The synchronization works in the opposite way for the gate "receive". The processes **Transmitter** and **Line** synchronize on gate "tout" only once the

timer for receiving an acknowledgment from process **Receiver** has timed out, in this case the same message would be retransmitted. (Note that this specification does not include a timer process).

```
specification Datalink [ get , give ] : noexit

LIBRARY Boolean ENDLIB

type sequenceNumber is Boolean (* modulo two *)
   sorts SeqNum
   opns  0      :                -> SeqNum
         inc    : SeqNum         -> SeqNum
         equal  : SeqNum,SeqNum  -> Bool

   eqns forall x , y : SeqNum
         ofsort SeqNum
  inc ( inc ( x ) ) = x ;
  ofsort Bool
  equal ( x , x ) = true ;
  equal ( 0 , inc ( x ) ) = false ;
  equal ( inc ( x ) , 0 ) = false ;
         equal ( inc ( x ) , inc ( y ) ) = equal ( x , y )
endtype

type BitString is Boolean   (* this is the data  *)
 sorts  BitString
 opns   empty   :                          -> BitString
        equal   : BitString , BitString    -> Bool
 eqns ofsort Bool forall x : BitString
               equal ( x , x ) = true
endtype

type FrameType is Boolean   (*  info or ack  *)
 sorts  FrameType
 opns   info, ack  :                       -> FrameType
        equal      : FrameType , FrameType -> Bool
 eqns ofsort Bool forall x : FrameType
        equal ( x , x )      = true ;
        equal ( ack , info ) = false ;
        equal ( info , ack ) = false ;
endtype

behaviour
   hide tout, send , receive in
     ( ( transmitter [ get, tout, send, receive ] (0)
       |||
         receiver [ give, send, receive ] (0)
       )
     |[ tout, send, receive ]|
         line [ tout, send, receive ]
     )

where

  process transmitter [ get, tout, send, receive ]
                           ( seq : SeqNum ) : noexit :=
      (* get the data *)
     get? data: BitString;
        (* send the data *)
```

```
56        sending [ tout, send, receive ] (seq, data)
57
58         (* ready to get the next set of data to be transmitted *)
59          >> transmitter [ get, tout, send, receive ] (inc(seq))
60
61    where
62      process sending [ tout, send, receive ]
63                              (seq: SeqNum, data:BitString) : exit :=
64
65           (* send the data to process line, associated with
66              the sequence number  *)
67
68        send !info !seq !data;
69
70           (* wait for an acknowledgment from process line *)
71          ( receive !ack !inc(seq) !empty; (* ack arrives *)
72              exit
73            []
74            tout; (* time out *)
75                  (* send the same set of data again *)
76              sending [ tout, send, receive ] ( seq, data )
77          )
78      endproc
79   endproc
80
81   process receiver [ give, send, receive ]
82                                    ( exp : SeqNum ) : noexit :=
83
84         (* receive the data from process line, associated
85            with the sequence number   *)
86     receive !info  ?rec:SeqNum ?data:BitString;
87
88           (* If the data received is the expected data *)
89       ( [ rec = exp ] ->
90                   (* send the data out *)
91            give !data;
92                   (* send an acknowledgment to process line *)
93               send !ack !inc(rec) !empty;
94                   (* ready to receive the next set of data *)
95                  receiver [ give, send, receive ] (inc(exp))
96
97         []  (* If the data received is not the expected *)
98         [ inc(rec) = exp ] ->
99            send !ack !inc(rec) !empty;
100              receiver [ give, send, receive ] (exp)
101      )
102  endproc
103
104  process line [ tout, send, receive ] : noexit :=
105
106        (* receive the data from process Transmitter if the
107           FrameType is info and send it to process Receiver.
108           Otherwise receive the acknowledgment from process
109           Receiver and send to process Transmitter if the
110           FrameType is ack  *)
111
112    send ?f:FrameType ?seq:SeqNum ?data:BitString;
113             (* send the receive data or acknowledgment *)
114        ( receive !f !seq !data;
```

```
115             line [ tout, send, receive ]
116          []
117          i;
118             tout; (* time out if any of the messages got lost *)
119                line [ tout, send, receive ]
120        )
121   endproc
122 endspec
```

The following is the parameterized tree of the above `Datalink` specification after the application of the reduction algorithm shown in chapter 4, 5:

```
tree<0>(0,0)
tree<0>(var_1,var_2:SeqNum)
     get ?BitString@1:BitString
        ==> tree<2>(var_1,BitString@1,var_2,info,var_1,BitString@1)
tree<2>(var_1:SeqNum; var_2:BitString; var_3:SeqNum; var_4:FrameType;
 var_5:SeqNum; var_6:BitString)
     | i (* timeout *) ==> tree<2>(var_1,var_2,var_3,info,var_5,var_6)
     | [(ack=var_4) and (inc(var_1)=var_5) and (empty=var_6)]
                    -> i ==> tree<0>(inc(var_1),var_3)  (* send-receive ack *)
     | [info=var_4] -> i   (* send-receive info *)
     | | [var_5=var_3] -> give !var_6  (* receive good *)
           ==> tree<2>(var_1,var_2,inc(var_3),ack,inc(var_5),empty)
     | | [inc(var_4)=var_5] -> i     (* receive bad *)
           ==> tree<2>(var_1,var_2,var_3,ack,inc(var_5),empty)
```

Because of the fact that gates `timeout`, `send` and `receive` are hidden, most of the actions are “internal” and therefore are removed by the reduction algorithm. The selection predicates associated with the internal actions have been transformed into guards, since LOTOS’s semantics does not allow the association of a selection predicate to an internal action. The selection predicates and guards associated to a removed internal action (due to the reduction) are added as a guard to the next action.

The logical flow of the parameterized tree shown above is illustrated in the following diagram, where internal actions are shown in square brackets (Figure 13).

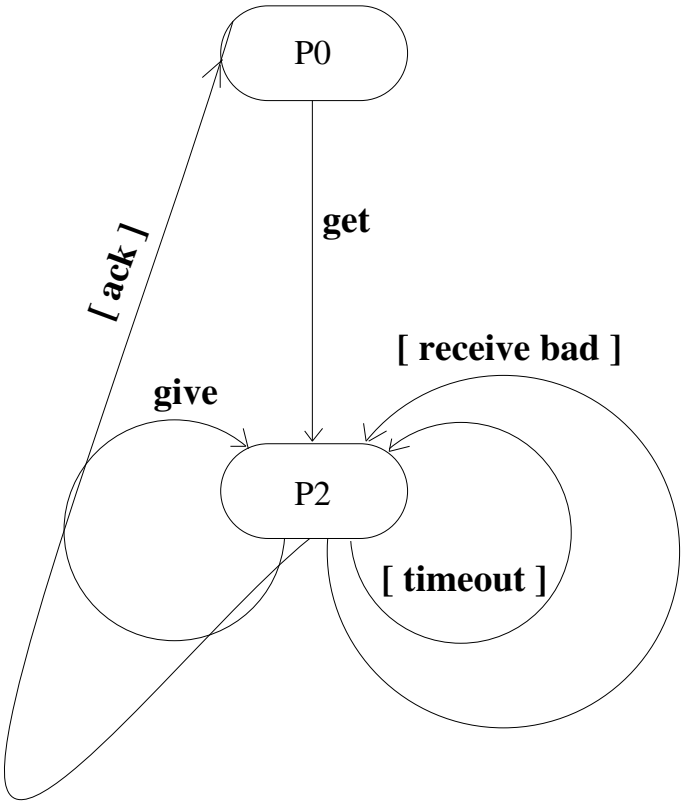


Figure 13 Diagram of specification *Datalink*

Note that this diagram connot be read without reference to the parameterized tree. For example, from the diagram above, it would appear that the sequence get, ack, get ... is possible. The parameterized tree, however, correctly shows that an ack cannot be sent immediately after a get. A give must be executed in order for the guards associated with the action of sending the ack to become true. Similarly, the guards make it impossible to execute two gives without an intermediate get. Most important is the role of the formal parameter `var_4`, which receives the

value info when tree<2> is instantiated to send or receive data, or receives the value ack when tree<2> is instantiated to send or receive information.

This parameterized tree describes the service provided by the Datalink specification, data is obtained from the user and given to the other user. Loops of internal actions correspond to the possibility that data is repeatedly lost. On the basis of this tree it is impossible to conclude that data obtained from one user will be eventually delivered to the other user, unless one includes a "fairness" assumption by which such loops are eventually exited.

Because of the fact that the tree describes the service and not the protocol, it is not appropriate for testing the protocol. In order to do that, we should remove the `hide` on the gates `send` and `receive`, and this will yield the following parameterized tree:

```
tree<0>(0,0)
tree<0>(var_1,var_2:SeqNum)
     get ?BitString@1:BitString
     | send !info !var_1 !var_2 ==> tree<2>(var_1,BitString@1,
                        var_2,info,var_1,BitString@1)
tree<2>(var_1:SeqNum; var_2:BitString; var_3:SeqNum; var_4:FrameType;
 var_5:SeqNum; var_6:BitString)
     | | i
     | | | i (hiding: tout)
     | | | | send !info !var_1 !var_2 ==> tree<2>(var_1,var_2,var_3,
                                        info,var_5,var_6)
     | | receive !ack !inc(var_1) !empty
           ([ack=var_4] and [inc(var_1)=var_5] and [empty=var_6] )
     | | | i ==> tree<0>(inc(var_1),var_3)
     | | receive !info !var_5 !var_6
           [info=var_4]
     | | | [var_5=var_3] -> give !var_6
     | | | | send !ack !inc(var_5) !empty ==> tree<2>(var_1,var_2,
                                    inc(var_3),ack,inc(var_5),empty)
     | | | [inc(var_4)=var_5] -> send !ack !inc(var_5) !empty  ==>
          tree<2>(var_1,var_2,var_3,ack,inc(var_5),empty)
```

The following is the monolithic representation of the specification:

```
 1 specification Datalink_mono[get,give,send,receive] : noexit
 2
 3 LIBRARY Boolean ENDLIB
 4
 5 type sequenceNumber is Boolean (* modulo two *)
 6    sorts SeqNum
 7    opns  0      :                -> SeqNum
 8          inc    : SeqNum         -> SeqNum
 9          equal  : SeqNum,SeqNum  -> Bool
10
```

```
11    eqns forall x , y : SeqNum
12          ofsort SeqNum
13   inc ( inc ( x ) ) = x ;
14   ofsort Bool
15   equal ( x , x ) = true ;
16   equal ( 0 , inc ( x ) ) = false ;
17   equal ( inc ( x ) , 0 ) = false ;
18          equal ( inc ( x ) , inc ( y ) ) = equal ( x , y )
19 endtype
20
21 type BitString is Boolean
22  sorts  BitString
23  opns   empty   :                            -> BitString
24         equal   : BitString , BitString      -> Bool
25  eqns ofsort Bool forall x : BitString
26                equal ( x , x ) = true
27 endtype
28
29 type FrameType is Boolean
30  sorts   FrameType
31  opns   info, ack  :                        -> FrameType
32         equal      : FrameType , FrameType -> Bool
33  eqns ofsort Bool forall x : FrameType
34         equal ( x , x )      = true ;
35         equal ( ack , info ) = false ;
36         equal ( info , ack ) = false ;
37 endtype
38
39 behaviour
40
41      P0[get,give,send,receive](0,0)
42 where
43   process  P0[get,give,send,receive](var_1,var_2:SeqNum):noexit :=
44      get ?BitString@1:BitString ;
45         send !info !var_1 !BitString@1;
46            P2[get,give,send,receive](var_1,BitString@1,var_2,info,
                                                    var_1,BitString@1)
47   endproc
48
49   process  P2[get,give,send,receive](var_1:SeqNum,var_2:BitString,
                                  var_3:SeqNum, var_4:FrameType,
                                  var_5:SeqNum, var_6:BitString):noexit :=
50
51      i ;
52         i  (* hiding: tout *);
53            send !info !var_1 !var_2 ;
54               P2[get,give,send,receive](var_1,var_2,var_3,info,
                                          var_5,var_6)
55      []
56
57      receive !ack !inc(var_1) !empty
58         [(ack = var_4) and (inc(var_1) = var_5) and (empty = var_6) ];
59            i  (* enable: exit *);
60               P0[get,give,send,receive](inc(var_1),var_3)
61      []
62
63      receive !info !var_5 !var_6 [info = var_4]  ;
64         ( [ var_5 = var_3 ] -> give !var_6 ;
65            send !ack !inc(var_5) !empty ;
```

```
66              P2[get,give,send,receive](var_1,var_2,inc(var_3),ack,
                                          inc(var_5),empty)
67          []
68
69          [inc(var_5) = var_3] -> send !ack !inc(var_5) !empty ;
70              P2[get,give,send,receive](var_1,var_2,var_3,ack,
                                              inc(var_5),empty)
71        )
72
73   endproc
74 endspec
```

The following is the monolithic representation of the symbolic tree after the application of the reduction algorithm shown in chapter 4, §5.

```
 1 specification Datalink_mono[get,give,send,receive] : noexit
 2
 3 LIBRARY Boolean ENDLIB
 4
 5 type sequenceNumber is Boolean (* modulo two *)
 6    sorts SeqNum
 7    opns  0      :                -> SeqNum
 8          inc    : SeqNum         -> SeqNum
 9          equal  : SeqNum,SeqNum  -> Bool
10
11    eqns forall x , y : SeqNum
12          ofsort SeqNum
13   inc ( inc ( x ) ) = x ;
14   ofsort Bool
15   equal ( x , x ) = true ;
16   equal ( 0 , inc ( x ) ) = false ;
17   equal ( inc ( x ) , 0 ) = false ;
18          equal ( inc ( x ) , inc ( y ) ) = equal ( x , y )
19 endtype
20
21 type BitString is Boolean
22  sorts  BitString
23  opns   empty   :                           -> BitString
24         equal   : BitString , BitString     -> Bool
25  eqns ofsort Bool forall x : BitString
26                equal ( x , x ) = true
27 endtype
28
29 type FrameType is Boolean
30  sorts   FrameType
31  opns   info, ack  :                       -> FrameType
32         equal      : FrameType , FrameType -> Bool
33  eqns ofsort Bool forall x : FrameType
34         equal ( x , x )      = true ;
35         equal ( ack , info ) = false ;
36         equal ( info , ack ) = false ;
37 endtype
38
39 behaviour
40
41      P0[get,give,send,receive](0,0)
42 where
43   process  P0[get,give,send,receive](var_1,var_2:SeqNum):noexit :=
44      get ?BitString@1:BitString ;
45         send !info !var_1 !BitString@1;
46            P2[get,give,send,receive](var_1,BitString@1,var_2,info,
                                        var_1,BitString@1)
47   endproc
48
49   process  P2[get,give,send,receive](var_1:SeqNum,var_2:BitString,
                                 var_3:SeqNum,var_4:FrameType,
                                 var_5:SeqNum, var_6:BitString):noexit :=
50
51      i ;
52            send !info !var_1 !var_2 ;
53               P2[get,give,send,receive](var_1,var_2,var_3,info,
```

```
                                          var_5,var_6)
54      []
55
56      receive !ack !inc(var_1) !empty
57         [(ack = var_4) and (inc(var_1) = var_5) and (empty = var_6) ];
58              P0[get,give,send,receive](inc(var_1),var_3)
59      []
60
61      receive !info !var_5 !var_6 [info = var_4]  ;
62         ( [ var_5 = var_3 ] -> give !var_6 ;
63            send !ack !inc(var_5) !empty ;
64               P2[get,give,send,receive](var_1,var_2,inc(var_3),ack,
                                          inc(var_5),empty)
65           []
66
67           [inc(var_5) = var_3] -> send !ack !inc(var_5) !empty ;
68               P2[get,give,send,receive](var_1,var_2,var_3,ack,
                                          inc(var_5),empty)
69         )
70
71   endproc
72 endspec
```

The internal actions at lines 52 and 59 were removed by the reduction algorithm since they do not affect the behaviour of the specification.

The three steps shown in chapter 5 will now be used in the generation of the test cases of the above specification (the reduced specification).

Since a process in the monolithic style specification represents a state, the above specification has two states (Figure 14).

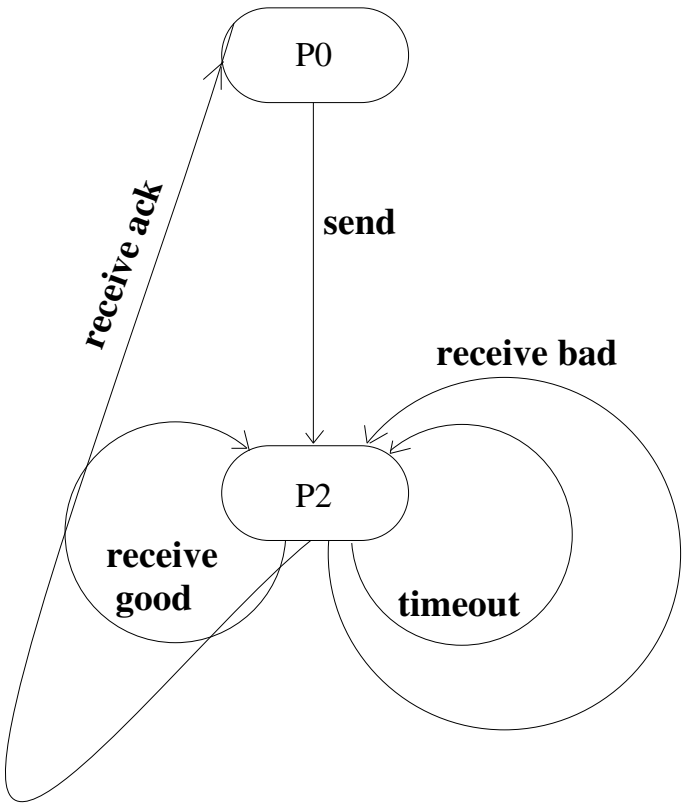


Figure 14 Diagram of specification *Datalink*

The labels used in the above diagram represent the following actions:

```
Labels          sequence of actions
send            get ?BitString@1;
                   send !info !var_2 !BitString@1;
                     P2[get,give,send,receive](var_1,BitString@1,var_2,
                                               info,var_1,BitString@1)
timeout         i ;
                   send !info !var_1 !var_2;
```

```
                    P2[get,give,send,receive](var_1,var_2,var_3,
                                              info,var_5,var_6)
receive ack     receive !ack !inc(var_1) !empty
                   [(ack = var_4) and (inc(var_1) = var_5)
                                  and (empty = var_6) ];
                     P0[get,give,send,receive](inc(var_1),var_3)
receive good    receive !info !var_5 !var_6 [info = var_4];
                   [ var_5 = var_3 ] -> give !var_6 ;
                       send !ack !inc(var_5) !empty ;
                         P2[get,give,send,receive](var_1,var_2,inc(var_3),
                                                   ack,inc(var_5),empty)
receive bad     receive !info !var_5 !var_6 [info = var_4];
                   [inc(var_5) = var_3] -> send !ack !inc(var_5) !empty ;
                      P2[get,give,send,receive](var_1,var_2,var_3,ack,
                                                inc(var_5),empty)
```

As in the case of the diagram of Fig 12, some of the transitions shown in Fig. 13 are possible only when the associated selection predicates are true. For example, the second alternative in process `P2` at line 56 (**receive ack**) cannot be executed in the first instantiation of process `P2` due to the selection predicate which depends on constants such as `ack`. Once the third alternative has been executed (line 61, **receive good**), process `P2` will reinstantiate itself by assigning the value `ack` to the fourth parameter `var_4`. Which makes the execution of the second alternative possible.

We now demonstrate the process of generating test cases for an implementation of the datalink service provider. According to the fact that both transmitter and receiver are specified, the test cases will test the global behaviour of the system, as it could be done by an observer capable of accessing the four gates get, give, send and receive.

One would think at first to use some standard graph coverage techniques in order to obtain test cases for the diagram of Fig. 13. This is not possible, however, because some of the transitions depend on guards and selection predicates, as mentioned above. Therefore, our test generation algorithm cannot be based on Fig. 13, but must be based on the monolithic representation of the symbolic tree. This implies that we do not know of any way of automating the process of test sequence generation carried out below, since this process depends on the knowledge of effects of tests and selection predicates on the execution of the

monolithic specification. By using a manual process, we can generate test cases as follow.

The following is the only initialization path of the second state `P2(var_1,var_2,var_3,var_4,var_5,var_6)`. `P0` does not have an initial path since the specification starts at state `P0`:

```
ip2
---
   get !BitString@1:BitString
      send ?info ?var_1 ?BitString@1
```

The test cases extracted from the specification in the evaluation step:

```
State P0(var_1,var_2)
--------
(1) get !BitString@1
       send ?info ?var_1 ?BitString@1
          + tr2
State P2(var_1,var_2,var_3,var_4,var_5,var_6)
--------
(2) ip2
       send ?info ?var_1 ?var_2
          + tr2
(3) ip2
       receive ?ack ?inc(var_1) ?empty
(4) ip2
       receive ?info ?var_5 ?var_6
          give ?var_6
             send ?ack ?inc(var_5) ?empty
                + tr2
(5) ip2
       receive ?info ?var_5 ?var_6
          send ?ack ?inc(var_5) ?empty
             + tr2
```

The last step in the test sequence generation is the termination phase. The termination path of state `P2` consists of the action

`receive ?ack ?inc(var_1) ?empty`. As it was mentioned above this action cannot be executed when the parameter `var_4` is not assigned to the constant `ack`.

The termination phase of state `P2(var_1,var_2,var_3,var_4,var_5,var_6)` is:

```
tr2
---
    receive ?info ?var_5 ?var_6
       give ?var_6
          send ?ack ?inc(var_5) ?empty
             receive ?ack ?inc(var_1) ?empty
```

The following is the complete test sequence of the specification.

```
State P0(0,0)
--------
(1) get !BitString@1
       send ?info ?0 ?BitString@1
          receive ?info ?0 ?BitString@1
             give ?BitString@1
                send ?ack ?inc(0) ?empty
                   receive ?ack ?inc(0) ?empty
State P2
--------
(2) get !BitString@1
       send ?info ?0 ?BitString@1
          send ?info ?0 ?BitString@1
            receive ?info ?0 ?BitString@1
               give ?BitString@1
                  send ?ack ?inc(0) ?empty
                     receive ?ack ?inc(0) ?empty
(3) get !BitString@1
       send ?info ?0 ?BitString@1
          receive ?info ?0 ?BitString@1
             give ?BitString@1
                send ?ack ?inc(0) ?empty
                   receive ?ack ?inc(0) ?empty
(4) get !BitString@1
```

```
        send ?info ?0 ?BitString@1
           receive ?info ?0 ?BitString@1
              give ?BitString@1
                 send ?ack ?inc(0) ?empty
                    receive ?ack ?inc(0) ?empty(5)
(5) get !BitString@1
        send ?info ?0 ?BitString@1
           receive ?info ?0 ?BitString@1
              send ?ack ?inc(0) ?empty
                 receive ?ack ?inc(0) ?empty
```

For the first three test cases of state `P2` (2,3,4), the specification was instantiated `P0(0,0)`. In the last test case (5), the specification was instantiated with `P0(0,inc(0))`, in order to satisfy the predicate assigned to it ([inc(var_5) = var_3]).

The third and fourth test cases would be deleted since they are similar to the first one. The remaining test cases are:

(1) which corresponds to the behaviour of the service provider in absence of error.

(2) which corresponds to the case of loss of data item, detected by timeout.

(5) which corresponds to the case of loss of acknowledgment or premature timeout, detected by reception of data item out of sequence.

The following are the test cases in TTCN form:

```
1
2  (1) get !BitString@1
3         send ?info ?0 ?BitString@1
4            receive ?info ?0 ?BitString@1
5               give ?BitString@1
6                  send ?ack ?inc(0) ?empty
7                     receive ?ack ?inc(0) ?empty     pass
8                     receive ?Otherwise              fail
9                     send ?Otherwise                 inconc
10                  send ?Otherwise                   fail
11               give ?Otherwise                      fail
12            receive ?Otherwise                      fail
```

```
13            send ?Otherwise                                inconc
14         send ?Otherwise                                   fail
15      ? Elapse TD                                          fail
16
17
18   (2) get !BitString@1
19         send ?info ?0 ?BitString@1
20            send ?info ?0 ?BitString@1
21               receive ?info ?0 ?BitString@1
22                  give ?BitString@1
23                     send ?ack ?inc(0) ?empty
24                        receive ?ack ?inc(0) ?empty pass
25                        receive ?Otherwise           fail
26                        send ?Otherwise              inconc
27                     send ?Otherwise                 fail
28                  give ?Otherwise                    fail
29               receive ?Otherwise                    fail
30               send ?Otherwise                       inconc
31            send ?Otherwise                          fail
32         send ?Otherwise                             fail
33      ? Elapse TD                                    fail
34
35
36   (3) get !BitString@1
37         send ?info ?0 ?BitString@1
38            receive ?info ?0 ?BitString@1
39               send ?ack ?inc(0) ?empty
40                  receive ?ack ?inc(0) ?empty       pass
41                  receive ?Otherwise                fail
42                  send ?Otherwise                   inconc
43               send ?Otherwise                      fail
44            receive ?Otherwise                      fail
45            send ?Otherwise                         inconc
46         send ?Otherwise                            fail
47      ? Elapse TD                                   fail
```

The alternative `send ?Otherwise` with the verdict `inconc` has been added, for example in lines 9 and 13 in the first test case, due to the internal action at line 51 in the reduced monolithic specification. This internal action can eliminate the

possibility of interacting through the gate `receive`. Intuitively, it corresponds to the case of timeout before the transmitter has received the acknowledgment.

Bibliography

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specifications, their Implementations and their Tests. In G. von Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing, and Verification IV*, pages 349–360. North-Holland, 1987.

[Feh87] M.C. Fehri. A System for Validating and Executing Lotos Data Abstractions (SVELDA). Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, November 1987.

[GL89] R. Guillemot and L. Logrippo. Derivation of Useful Execution Trees from LOTOS by using an Interpreter. In K. J. Turner, editor, *Formal Description Techniques*, pages 311–325. North-Holland, 1989.

[Gue89] D. Gueraichi. Derivation of Test Cases for LAP-B from a Formal Specification in LOTOS. Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, 1989.

[HH88] M. Haj-Hussein. An Interactive System for LOTOS Applications (ISLA). Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, November 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, UK, LTD., 1985.

[Int88] International Organization for Standardization. *Information Processing Systems - Open Systems Interconnection - LOTOS IS8807 - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, June 1988.

[Int89] International Organization for Standardization. *Information Retrieval, Transfer and Management for Open Systems Interconnection - The Tree and Tabular Combined Notation (TTCN)*, February 1989.

[Int92] International Organization for Standardization. *Formal Methods in Conformance Testing - Working Draft*, May 1992.

[MF91] O.Bellal M.Dubuc, G.v. Bochmann and F.Saba. Translation from TTCN to LOTOS and the Validation of Test Cases. In

E. Vazquez (Eds.) J. Quemada, J.Manas, editor, *Formal Description Technique, III*, pages 141–155. North-Holland, 1991.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Sringer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Lecture Notes in Computer Science. Prentice-Hall, 1989.

[QFP88] J. Quemada, A. Fernandez, and S. Pavon. Transforming LOTOS specifications with LOLA. the parameterized expansion. In K.J. Turner, editor, *Proceedings of the first National Conference on Formal Description Techniques (FORTE)*, pages 45–54. North-Holland, 1988.

[Ura87] H. Ural. A test derivation method for protocol conformance testing. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification VII*, pages 347–358. North-Holland, 1987.

[vE89] P.H.J. van Eijk. The Design of a Simulator Tool. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 351–390. North-Holland, 1989.

[VSvS88] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 189–204. North-Holland, 1988.