An Interactive System for LOTOS Applications

(ISLA)

by

Mazen Haj-Hussein

THESIS SUBMITTED

TO THE SCHOOL OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

MASTER OF COMPUTER SCIENCE DEGREE

at the

UNIVERSITY OF OTTAWA

August 1988

ABSTRACT

LOTOS is a Formal Description Technique developed to specify protocols and services for the Open Systems Interconnection. It is expected to become an ISO standard in the very near future.

The topic of the thesis is the development and the implementation of an interpreter for LOTOS specification behaviours called ISLA (Interactive System for LOTOS Applications). ISLA is the central part of the LOTOS interpreter developed by the Protocols Research Group of the University of Ottawa.

ISLA allows users to simulate the execution of a specification and check whether it behaves correctly. The simulation can proceed either under user guidance, by directing the execution step-by-step, or by symbolic execution. The thesis includes the description of the most important features of ISLA that help to obtain an analysis of the specification under simulation.

The thesis is structured as follows: In chapter 1, an overview of the thesis motivation and of the methodology is introduced, and the overall structure of the University of Ottawa LOTOS Interpreter is presented. The next chapter covers the syntax and semantics of LOTOS. In chapter 3, a detailed description of ISLA's functionality is given. Chapter 4 describes the implementation of important functions supported by ISLA. Chapter 5 reports on our experiences in simulating a real system: the OSI Transport Service provider. Proposed tracing methodologies are also presented. Finally, the conclusion of the thesis follows in chapter 6.

A table of the syntax of LOTOS behaviour constructs is given in Appendix A. A LOTOS specification is constructed in Appendix B. In Appendix C, we demonstrate the user interface of ISLA by a session log of a simulation. The commands supported by ISLA, together with their descriptions, are listed in Appendix D. The Internal Form syntax of LOTOS specifications is presented in Appendix E. Finally, Appendix F includes the implementation of the inference rules in Prolog.

Acknowledgments

I am most grateful to my supervisor, Dr. Luigi Logrippo, for all the guidance and advice that he has given me throughout my graduate studies.

I acknowledge the contributions made by the University of Ottawa LOTOS group; in particular A. Obaid for his many useful discussions. The compiler part was written by J.P. Briand and the abstract data type interpreter was implemented by M.C. Fehri. B.Stepien and R. Guillemot also collaborated in the programming. Useful comments were also given by J. Sincennes. In addition, the LOTOS group at Twente University have been generous in sharing their experiences with our group.

I would like to thank the Natural Sciences and Engineering Research Council for a scholarship and additional funding. This work was also supported in part by Bell-Northern Research.

Finally, I would like to express my deepest thanks to my wife, Allyson, for her patience and support.

Table of Contents

Chapter 1: Introduction

The International Organization for Standardization (ISO) has been working on the development of international standards for Open Systems Interconnection (OSI). The purpose of OSI is to provide common basis standards, meant to be implemented in compatible ways across the world, enabling heterogeneous computer systems to communicate meaningfully with each other. These standards include a framework, called OSI Reference Model, the services offered by each layer of that model, and the protocols that are needed to provide those services.

All these standard protocols and services are currently specified by informal methods, consisting of natural languages and various semi formal notations. This could easily lead to misinterpretation of the standards by some implementors, thus resulting in incompatible implementations. Furthermore, this makes it impossible to carry out rigorous analysis of the standards at the specification level. For these reasons ISO has recognized the need of Formal Description Techniques (FDTs) to provide the basis of OSI standards.

The main objectives of such FDTs are to make it possible to produce unambiguous, precise, and complete standard specifications, and to provide a formally well-defined basis for the verification and validation of the standards prior to implementation, as well as conformance testing after implementation.

LOTOS (Language Of Temporal Ordering Specification) is an FDT being standardized within ISO for formally specifying protocols and services of the Open Systems Interconnection [OSI1]. LOTOS is also applicable to distributed systems beyond OSI such as telephone switching systems.

LOTOS has two components: a ’data’ component that deals with the description of data structures and value expressions based on the formal theory of Abstract Data Types ACT ONE [JDM], and a ’control’ component that describes the externally observable *behaviour* of the system. This component is based on Milner’s CCS (Calculus of Communicating Systems) [MIL] and Hoare’s CSP (Communicating Sequential Processes) [Hoare].

One of the most important features of LOTOS is the fact that thedynamic semantics of a specification’s behaviour are precisely defined in terms of inference rules that make it possible to determine its meaning unambiguously [ISO1]. Transforming these rules into equivalent executable rules makes LOTOS an executable FDT.

1.1 Motivation of the Thesis

Some questions of principle could be asked about writing an interpreter for a specification language such as LOTOS. Specification languages, unlike implementation languages, do not have to be executable, although a methodology for obtaining an implementation from a specification is highly desirable. LOTOS’ philosophy involves executability, even if it is possible to obtain LOTOS constructs that are non-executable, or executable only very inefficiently (such constructs would have to be modified in order to be able to run the specifications containing them with our interpreter). The advantages of being able to execute specifications, although possibly inefficiently, are several. One is the fact that the behaviour of the specification can be checked with respect to the intended behaviour of the entity. Other applications involve test sequence generation, etc [URS].

The main motivation behind this thesis is the development of a tool to prototype the dynamic behaviour of LOTOS specifications.

The tool features some useful characteristics, such as:

a) portability: the tool is implemented in an environment that is widely used in research establishments, so that it could be used on most popular systems.

b) modularity: the tool is well modularized such that functionalities can be easily modified, added, or tested.

The basic operation of such a tool is to allow step-by-step execution of LOTOS specifications. At each step, all possible actions are offered and the user can choose the action (s)he wants to execute next. In this way, the user can check whether the actions conform to the ones expected by the designer of the specification.

This tool can also be used to study certain protocols in detail. For example, to determine if a given sequence of interactions is acceptable by the protocol, what exactly can happen at a given point, etc. It can also be used as a tutorial
tool for teaching LOTOS, to show how a given specification behaves.

1.2 The University of Ottawa LOTOS Interpreter

The Protocols Research Group of the University of Ottawa has developed various 'LOTOS-Based' interpreters for the last three years.

The current interpreter is based solely on the specification of LOTOS. The structure of the interpreter is shown in figure 1.1. It corresponds to the structure described in [LOBF].

First, the LOTOS specification source is checked for its syntax and static semantics according to [ISO1], 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 compiler.

The real LOTOS interpreter runs on the internal representation of the data and control components. It consists of two interpreters: The **ADT Interpreter** which validates and evaluates value expressions. This interpreter is called **SVELDA** (System for Validating and Executing LOTOS Data Abstractions) [FEH]. And 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.

The construction of the **ISLA** was done by the following steps:

Step 1. Mapping the definition of the inference rules of LOTOS
dynamic behaviours to a constructive definition.

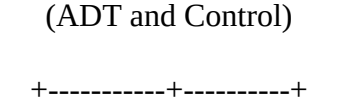
Step 2. Transforming the latter inference rules into equivalent
executable rules in a systematic way.

Step 3. Rewriting the latter inference rules in such a way that
their function will be realised more efficiently, by
preserving their correctness.

Step 4. Developing functions that help analyzing important
aspects of the dynamic behaviour of LOTOS specifications.

Step 5. Develop a user interface, enabling users to perform the
above functions on a given specification.

The above five steps is the work of the author. The other elements of the interpreter were implemented by other members of the group.



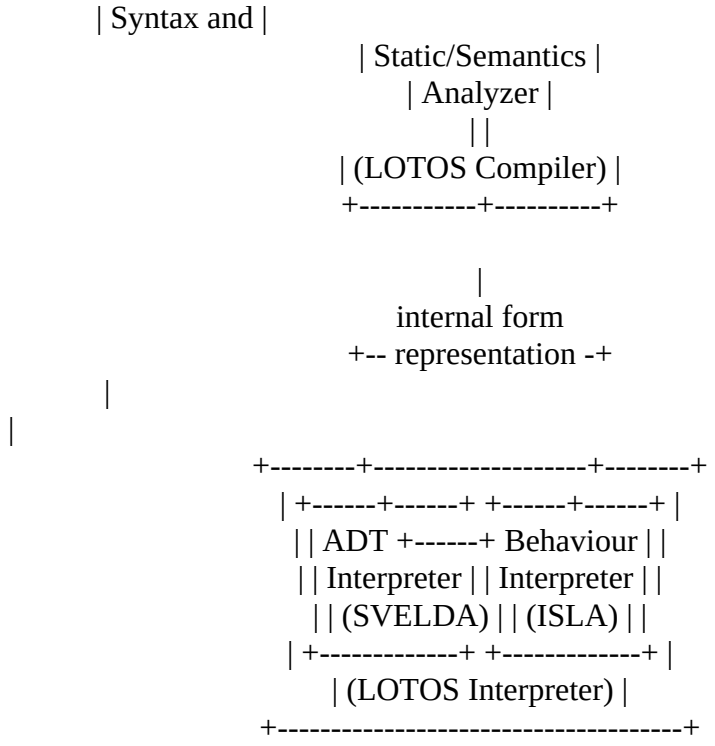


Figure 1.1 The LOTOS Interpreter Structure.

1.3 Previous Work

The first 'LOTOS-Based' interpreter was developed by Jean-Pierre Briand and Abdel Obaid, members of the Protocols Research Group of the University of Ottawa. The interpreter was called SINAPS (for Simulation and INference rules APplication System) [OBA2]. SINAPS was based on the theory of CCS and was developed as an experimental tool, rather than a production tool able to undertake large jobs. Obaid was the first to show how the semantics of LOTOS, defined in terms of inference rules, could be programmed in Prolog [OBA1].

In [PAP] a step-by-step simulator for the specification language ECCS (Extended CCS) is presented. This tool was programmed in Prolog.

A trace generator for CSP, also written in Prolog, is described in [KOR] where all traces up to, but not including, terminationor recursion can be generated.

Very recently, a LOTOS simulator called HIPPO was developed in ESPRIT/SEDOS (Software Environment for the Design of Open Systems) project. This work was sponsored by the Commission of the European Community and it was based on the simulator tool discussed in details by Peter van Eijk in his Ph.D. thesis [EIJ]. HIPPO also consists of a compiler and an interpreter where both are programmed in C. The main advantage of programming the interpreter part in C with respect to Prolog is that it is more efficient. However, we have found that Prolog was also a very appropriate language for such a tool, because of the following reasons:

1) LOTOS operational semantics can be specified in a natural way
using Prolog clauses.
2) via backtracking in Prolog, all solutions can be generated for
one goal.
3) using unification, a copy of a Prolog structure can be
obtained and direct substitutions can be performed easily,
where these operations are often used during simulation.

4) since Prolog is an interactive language, any functionality of
the interpreter can be easily tested, removed or added.

Using an optimized definition of the derivation semantics, discussed in chapter 4, the execution efficiency of the interpreter is quite adequate even when prototyping large LOTOS specifications (chapter 5).

ISLA and HIPPO have similar functionalities, although the current version of HIPPO does not support symbolic execution, discussed in chapter 3. We also found that ISLA has more execution trace support, since the user can trace, for example, the application of the inference rules on a behaviour expression, the evaluation of predicates, etc. ISLA can also help the user to find proper values for an action taken from user predefined databases, as discussed in chapter 3.

Chapter 2: Overview of LOTOS

2.1 Introduction

In LOTOS, interprocess communication occurs by means of a "rendez-vous" mechanism, called "interaction". Processes participate in a "rendez-vous" if and only if they all offer an event at the same interaction point, called "gate", with matching information. This can range from a specific value to a type only. For instance, a "rendez-vous" can occur when one process is expecting a value of a certain type on a specific gate and the other offers a value of that type on the same gate. Thus, the concept of "information flow" strictly does not exist in LOTOS, unlike in most programming languages, where messages are directed from a process specifically to another named process.

We recall that the LOTOS language has two main components: the *data type component* based on algebraic "Abstract Data Types" (ADT) specification, as in ACT ONE , and the *control component* based on Milner's CCS and Hoare's CSP.

2.2 Data Type Component

LOTOS adopted the Abstract Data Type language ACT-ONE for defining its data types such as values, value expressions, data structures and operations on them. The choice of *abstract* data types for LOTOS, as opposed to *concrete* data types, is consistent with the requirement of abstraction from implementation details. Abstract types do not indicate how data values are actually represented and manipulated in memory, but only define the essential properties of data and operations that any correct implementation is required to satisfy. Since ACT-ONE allows "nonconstructive" specifications, the execution may end up into infinite loops or deadlocks. Detecting and/or repairing such specifications may be impossible [FEH].

ACT-ONE is an algebraic specification method to write unparameterized and parameterized specifications. ACT-ONE has the following features:

1- Reference to already defined specifications in a library.

2- Combination, renaming and parameterization of specifications.

3- Actualization of parameterized specifications.

4- Extension of specifications with operations and sorts.

The basic form of a data type specification consists of a *signature*, that gives all the information required to build syntactically correct terms, (also called value expressions), and, possibly, a list of *equations*.

A *signature* includes the definition of data carriers and operations. The declaration of an operation will include its *domain*, which consists of a list of zero or more sorts, and its *range*, which consists of exactly one sort. The following example is a type definition of the natural numbers. It consists only of a signature that involves a single sort 'Nat', and the operations '0' and 'Succ'. The operation 'Succ' can be applied to single elements of sort 'Nat', resulting in another element of sort 'Nat', which is indicated by 'Nat -> Nat'. The operation '0' is an operation with no arguments which results in an element of sort 'Nat', as is indicated by the notation '-> Nat'. The definition is named 'Natural-Numbers', so that it may be referred to by other definitions, and combined with them.

```
type Natural-Numbers
        sorts Nat
        opns 0 :-> Nat
             Succ: Nat -> Nat
```

endtype

For example, 'Succ(Succ(0))' is a term of sort 'Nat'.

Suppose now that we want to define the prefix operation 'largest':

largest : Nat,Nat -> Nat

which takes two arguments of type 'Nat' and returns the largest of them. For example, 'largest(Succ(Succ(0)), Succ(0))' will yield 'Succ(Succ(0))'. We need a new construct, called *equation* to express properties of operations. The purpose of an equation is to state that two syntactically different terms denote the same value. A correct definition of the properties of the 'largest' operation is:

eqns forall X :Nat ,Y: Nat
ofsort Nat
largest(0,X) = X;
largest(X,0) = X;
largest(Succ(X),Succ(Y)) = Succ(largest(X,Y));

where the first and the second equations state that the largest of any natural number X and the natural number 0 is X. In addition, the largest of two non-zero natural numbers, denoted by 'Succ(X)' and 'Succ(Y)', is defined to be 'Succ(largest(X,Y))', where largest(X,Y) has to be evaluated. By induction on the structure of terms, and by using these equations, it can be easily proved that any term containing one or more 'largest' operations can be expressed by a term containing only '0' and 'Succ' operations.

The complete definition of natural numbers with the 'largest' operation is:

type Natural-Numbers
sorts Nat
opns 0 :-> Nat
Succ: Nat -> Nat
eqns
forall X :Nat ,Y: Nat
ofsort Nat
largest(0,X) = X;
largest(X,0) = X;
largest(Succ(X),Succ(Y)) = Succ(largest(X,Y));
endtype

The **ADT Interpreter** evaluates (or rewrites) a given term using the internal form representation of the abstract data type equations, where they are oriented as rewriting rules [FEH]. ISLA calls the latter evaluator whenever a value expression needs to be evaluated. The evaluator function is called 'eval'. It returns the evaluation of a given term. For example 'eval(largest(Succ(0),0))' returns the value 'Succ(0)' with respect to the above equation definitions.

The data type component is outside the scope of this thesis. We refer readers to [FEH] for more details.

In the examples included in this thesis, we often use operators such as "+", "<", ">", "mod" etc, that require further definition in the abstract data type components. For brevity we do not provide these definitions, rather we assume that these operators behave according to their intuitive meaning.

2.3 Control Component

The control component of LOTOS describes the behaviour of a specification via a hierarchy of process definitions. In the following sections, we describe the elements of this component.

2.3.1 Processes

A Specification in LOTOS can be seen as a process that may consist of interacting subprocesses. Each subprocess may in turn consist of other subprocesses. Each process can be imagined as a black box that is capable of interacting with other processes (its environment) via common interaction points called "gates". A process can also perform internal, unobservable actions denoted by '**i**'. The environment of a process consists of the other processes in the specification, plus an unspecified process which is always ready to interact at any gate. In the interpreter, the user takes the role of this process.

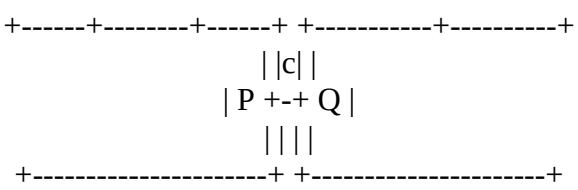


Figure 2.1 Two interacting processes

Figure 2.1 shows two processes P and Q interacting among themselves and the environment at gate **c**.

The syntax of a LOTOS *specification* is of the form:
specification spec-name[g1,..,gn](v1,..,vm):functionality
behaviour
<behaviour expression>
where
<process definitions>
endspec

The syntax of a *process definition* is of the form:
process proc-name[g1,..,gn](v1,..,vm):functionality :=
<behaviour expression>
where
<process definitions>
endproc
where the syntax of a *behaviour expression* is described in table 2.2 .
Therefore the definition of Figure 2.1 will be of the form:
process P[a,b,c] := <behaviour expression> **endproc.**
process Q[c,d] := <behaviour expression> **endproc.**

2.3.2 Actions

The basic element of a process behaviour is the action which represents a *synchronization* between processes. An action consists of a gate name (interaction point), a list of events, and an optional predicate that restricts the event values to those satisfying the predicate.

An event can be either of the form **!E**, denoting the offering of the value **E**, or of the form **?x:s**, denoting the readiness to accept any value of sort **s**. Further conditions on the values that can be accepted can be expressed by "selection predicates" in square brackets. For example:

g ?x:Nat !Succ(0) [x > Succ(Succ(0))]

is an observable LOTOS action which occurs at gate **g** and expects from the environment

a value for **x** of sort **Nat** restricted to be greater than two, while at the same time offering the value one.

As mentioned above, there is another type of action in LOTOS denoted by '**i**', which is an *internal action*. It is also called "unobservable action" because it does not interact with the environment.

2.3.3 Types of Interactions

Interprocess communication in LOTOS occurs when two or more processes, having a "rendez-vous" on a gate, agree on a value to be established. This is the case of *matching actions*.

Table 2.1 shows all possible types of interactions between two processes. When more than two processes are involved, similar rules apply. We recall that **eval(E)** indicates the value of the expression **E**

process process sync. interaction effect					
A	B	condition	sort		
$g!E_1$	$g!E_2$	$eval(E_1)$	value	synchronization	
		=	matching		
		$eval(E_2)$			
g!E	g?x:t	eval(E)	value	after	
	∈		passing	synchronization	
	domain(t)		x = eval(E)		
$g?x:t_1$	$g?y:t_2$	$t_1 = t_2$	value	after	
			generation	synchronization	
			x = y = v		
			$v \in domain(t_1)$		

Table 2.1 Types of Interactions

Example:

Process A is prepared to accept a natural number 4,5,6,7,8,9, or 10 at the gate g, as denoted by the following action:

g?X:Nat [X > 3 and X < 11]

and at the same time Process B is ready to accept a multiple of 3 natural number at gate g, as denoted by the action:

g?X:Nat [X mod 3 = 0]

then an interaction will occur at gate **g**, if the environment cooperates by offering a natural number satisfying the conditions of the above actions, namely 6 or 9.

2.3.4 Behaviour Expressions

In this section, we represent the semantics of LOTOS behaviour expressions. For clarity, in the examples where the events on the actions do not affect the semantic of the behaviour, only the gates of the actions are given.

The syntax of LOTOS behaviour expressions is shown in table 2.2.

| LOTOS behaviour expressions |
+---+---+
g d_1 ... d_n[P]; B	action prefix
where	
d_i = !t_i or **?x_i:s_i**	
+---+---+
| **i; B** | internal action prefix |
+---+---+
exit	successful termination
or	
exit(E_1,..., E_n)	
where	
E_i is a term or	
E_i = **any s_i**	
+---+---+
| **let x_1=t_1,..,x_n=t_n in B** | local definition |
+---+---|
| **choice $g_=$ in** [g_1,..,g_n] [] B | summation on gates |
+---+---+
| **choice x:s [] B** | summation on values |
+---+---+
| **par g in [g_1,..,g_n] op B** | par |
+---+---+
| **hide g_1,..,g_n in B** | hiding |
+---+---+
| **B1 >> accept** | enable |
| **x_1:s_1,..,x_n:s_n in B2** | |
+---+---+
| **B1 [> B2** | disable |
+---+---+
	parallel
B1 ||| B2	(interleaving)
B1 || B2	(full synchronization)
B1 |[g_1,..,g_n]| B2	(selected synchronization)
+---+---+
| **B1 [] B2** | choice |
+---+---+
| **[Guard] -> B** | guarded |
+---+---+
| **stop** | deadlock (inaction) |
+---+---+
| **(B)** | nested |

| **p[g_1,..,g_n](t_1,..,t_n)** | process instantiation |

Table 2.2 LOTOS Syntax

2.3.4.1 Inaction

B = **stop** : B cannot offer any action to the environment nor can it perform internal events. This behaviour is also denoted **deadlock**.

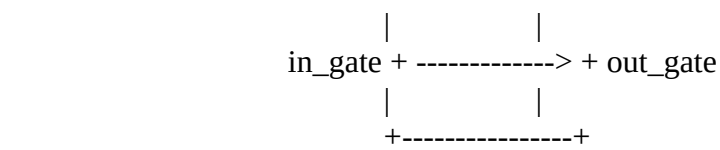
2.3.4.2 Action Prefix

B = **g d_1 ... d_n[P]; B2** : The behaviour B offers participation in event **g d_1 ... d_n[P]**; if the interaction occurs the resulting behaviour is given by **B2**.

d_i is either **!E**, denoting the offering of the value eval(**E**), which is the value of expression **E**, or **?x:s**, denoting that a value for the variable **x** of sort **s** is expected.

EXAMPLE

```
process in-out-buffer[in-gate,out-gate] :=
        in-gate ?x:Nat [x>0];
        out-gate!x;
        stop
    endproc
```



This process accepts a positive natural number at gate in-gate, offers the same value at gate out-gate, and then stops.

2.3.4.3 Internal Action Prefix

B = **i; B2** : The behaviour B may perform nondeterministically an 'unobservable' internal action, denoted by **i**, and transforms into **B2**. This internal action is due to an internal event such as disconnection events, shutdown, or internal synchronization between processes. This action is not observed by the environment.

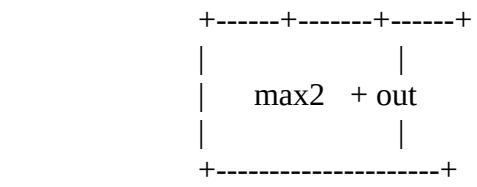
2.3.4.4 Choice

B = **B1 [] B2** : Depending on the action offered by theenvironment, B will behave as **B1** or **B2**.

EXAMPLE

```
process max2[in1,in2,out] :=
        in1?x:Nat;
        in2?y:Nat;
        out!largest(x,y);
        stop
    []
in2?x:Nat;
        in1?y:Nat;
        out!largest(x,y);
        stop
```

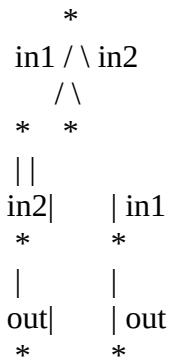
endproc



This process accepts two natural numbers at gates **in1** and **in2** in any order and offers the largest of these numbers at gate **out**. The accepted sequence of actions by this process with respect to the interaction gate names, are: **in1; in2; out** and **in2; in1; out**.

The accepted sequences of a given behaviour can be represented by a tree-like structure called *Behaviour Tree* or BT, where the root of the tree is the initial state of the behaviour, and the edges of each node represent the gates of the possible next actions.

The Behaviour Tree of the above example is:



2.3.4.5 Guarded Behaviour

B = **[Guard] -> B1** : B will behave as **B1** if Guard is evaluated to **true**, and behaves as **stop** otherwise.

EXAMPLE

[x > 0] -> g1!(x+3); **stop**

[]

[x < 0] -> g2!x; **stop**

If x = 3 then the above behaviour is equivalent to

g1!6; **stop**

If x = -3 then the above behaviour is equivalent to

g2!-3; **stop**

If x = 0 then the above behaviour is equivalent to

stop

2.3.4.6 Process Instantiation

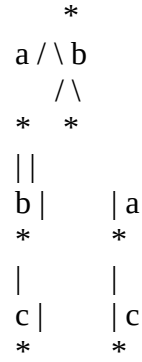
B = **P[g_1,..,g_n](t_1,..,t_n)** : B behaves as the behaviour of process definition **P**, with the substitution of the formal variable parameters by **t_1,..,t_n** and with the formal gates relabelled by the actual gates **g_1,..,g_n**.

EXAMPLE

The Behaviour Tree of the following call:

max2[a,b,c]

where process **max2** is as defined above, is:



In LOTOS, infinite behaviour can be defined using recursive process instantiation.

EXAMPLE

```
process in-out-buffer[in-gate,out-gate] :=
        in-gate ?x:Nat [x>0];
        out-gate!x;
        in-out-buffer[in-gate,out-gate]
    endproc
```

The call **in-out-buffer[in-gate,out-gate]** accepts the sequence:

in-gate out-gate in-gate out-gate in-gate ....

EXAMPLE

```
process in-out-buffer[in-gate,out-gate] :=
        in-gate ?x:Nat [x>0];
        out-gate!x;
        in-out-buffer[out-gate,in-gate]
    endproc
```

The call **in-out-buffer[a,b]** accepts the infinite sequence:

a b b a a b b a a ...

EXAMPLE

The following process can accept N sequential messages at the gate AP.

```
process receive-N-messages[AP](N:Nat) :=
        [N > 0] -> AP?Mes:M;
                receive-N-messages[AP](N-1)
    endproc
```

2.3.4.7 Local Definition

B = **let** $x_1=t_1,..,x_n=t_n$ **in B2** : All the instances of $x_1,..,x_n$ in **B2** are replaced by $t_1,..,t_n$ respectively.

EXAMPLE

```
let x:Nat = Succ(0), y:Bool = true in
        g1!x; g2!y; stop
```

This behaviour is equivalent to

```
        g1!Succ(0); g2!true; stop
```

2.3.4.8 Summation on Gates

B = choice g in [g_1,..,g_n] [] B2 : B is a shorthand for the behaviour **(B2)[g_1/g] [] ... [] (B2)[g_n/g]**, where **(B)[g_i/g]** denotes that **g** is relabelled by **g_i** for every action that may be performed by **B** on gate **g**.

EXAMPLE

choice a **in** [b,c] **[]**

a?x:Nat; d!(x+1); **stop**

is equivalent to

b?x:Nat; d!(x+1); **stop**

[]

c?x:Nat; d!(x+1); **stop**

2.3.4.9 Summation on Values

B = **choice x:s [] B2** : B is equivalent to [t1/x]B2 [] ... [][tn/x]B2, where t1,..,tn are all possible value expressions of sort s.

EXAMPLE

choice x:Nat **[]**

[x mod 2 = 0] -> g!x; **stop**

is equivalent to

g!0; **stop**

[]

g!2; **stop**

[]

g!4; **stop**

[]

g!6; **stop**

[]

.

.

.

2.3.4.10 Parallel Composition

B = **B1 |[g_1,..,g_n]| B2** : **B1** and **B2** are executed independently, except for the actions at any of the gates **g_1,..,g_n**, where **B1** and **B2** must synchronize.

Note that:

- ||| is equivalent to **|[]|**
- || is equivalent to **|[g_1,..,g_m]|** where **g_1,..,g_m** are all possible gates of both behaviours.

EXAMPLE

(a;b;c;stop) |[b]| (b;d;stop)

This behaviour accepts the following sequence of interactions:

a b c d

a b d c

The Behaviour Tree is

*****

a|

*

b|

*

c / \ d

/ \

* *

d| | c

* *

EXAMPLE

(a;b;c;stop) || (a;b;c;d;stop)

This behaviour accepts:

a b c

EXAMPLE

(a;b;stop) ||| (c;d;stop)

This behaviour accepts:

a b c d

a c b d

a c d b

c d a b

c a b d

c a d b

2.3.4.11 Hiding

B = **hide g_1,..,g_n in B2** : any action that may be performed by **B2** on gates **g_1,..,g_n** becomes an internal action. That is, the environment of **B2** cannot participate in these actions.

EXAMPLE

hide b in (a;b;c;stop) |[b]| (b;d;stop)

The sequences of actions that this behaviour may perform are:

a i c d

a i d c

which are externally observable as:

a c d

a d c

2.3.4.12 par-Expression

B = **par g in [g_1,..,g_n] op B2** : B is a shorthand of the behaviour **(B2)[g_1/g] op .. op (B2)[g_n/g]**.

EXAMPLE

par a in [b,c] ||

a; d;stop

is equivalent to

b; d; stop

||

c; d; stop

2.3.4.13 Successful Termination

B = **exit(E_1,..,E_n)** : denotes the successful termination of the behaviour B. The parameters **E_1,..,E_n** are the results of B and will be offered in an action performed on a specific gate denoted by d . The resulting behaviour of B is **stop**.

EXAMPLE

a?x:Nat; b?y:Bool; exit(x,y)

The above behaviour expression accepts values for **x** and **y** at the gates **a** and **b** respectively, then it offers **x** and **y** at gate d and stops.

2.3.4.14 Disable

B = **B1 [> B2** : B performs as **B1**, and during the life of **B1**, **B2** can disable **B1** then start executing, unless **B1** is successfully terminated. Note that **B1** can be disabled before any of its actions is executed.

The disabling operator is useful to express situations where aprocess can be interrupted by an exceptional event during normal functioning. Such situations are common in communication systems, the most obvious case being a disconnection.

EXAMPLE

a;b;exit [> c;d;stop

This behaviour accepts the following sequences:

c d

a c d

a b c d

a b δ

d denote the event performed by **exit**.

2.3.4.15 Sequential Composition (enable)

B = **B1 >> B2** : B performs as **B1** until successfully terminates, then it performs as **B2**.

EXAMPLE

(a;b;c;exit [] d;e;stop) >> f;g;stop

This behaviour accepts the following interaction sequences:

a b c f g

d e

In the first sequence, there exists an internal event between the actions **c** and **f**. This internal event is due to the successful termination of **B1** denoted by **exit**.

2.3.4.16 Sequential Composition with Value Passing (enable)

B = **B1 >> accept** $x_1:s_1,..,x_n:s_n$ **in B2** : B performs as **B1** until successfully terminates, then it performs as **B2** taking the results of **B1** in the variables $x_1:s_1,..,x_n:s_n$ in **B2**.

The number and sorts of the values that are passed at the successful termination of **B1** must be comparable with the value declarations in the **accept** statement.

The sequential composition with value passing is useful when **B2** depends on parameters generated by **B1**.

EXAMPLE

```
Connection-Phase[AP1,AP2] >>
     accept EXP-DATA:Bool,QofS:Nat in
        Data-Phase[AP1,AP2](EXP-DATA,QofS)
```

In the above example, two parameters determined in the Connection-Phase are passed to the Data-Phase: the expedited data option that indicates whether expedited data can be transferred or not, and the quality of service of the connection during the data phase.

2.3.4.17 Relabelling

B = **(B')**$[g_1/h_1,..,g_n/h_n]$: the gate h_i is relabelled by g_i for every action that **B'** may

perform on the gate $\mathbf{h_i}$.

EXAMPLE

(a;b;a;stop)[c/a,e/b]

This behaviour accepts the following sequence:

c e c

The relabelling behaviour expression is *not* in the syntax of LOTOS. It is constructed dynamically as an additional expression (e.g. when a process instantiation is to be executed).

Note that the relabelling is performed after deriving the actions that are offered by the relabelled behaviour, and it cannot be interpreted as performed statically by simply changing the gate names in the relabelled behaviour then deriving the actions, because in the second way, the semantics of the behaviour may change.

For example, let:

B = **(a;c;stop |[a]| b;stop)**

then B cannot offer the action **a** because it can not synchronize with an action offered by the behaviour **b;stop**. Therefore the action **b** is the only action that can be offered by B then it stops.

Now suppose we have the following behaviour:

B2 = **(B)[a/b]**

where B is the same as above. Then B2 can offer the action **a**. This is done by first obtaining all the actions offered by B, (only action **b**) then relabelling the actions with respect to the relabelling list (**b** relabelled by **a**).

If the gates were relabelled before deriving the actions then we would obtain the following behaviour:

B3 = **(a;c;stop |[a]| a;stop)**

In this case, B3 could offer the action **a** due to a synchronization, where this was not possible in B. Action **c** would then follow.

We conclude that

B2 = **(a;c;stop |[a]| b;stop)[a/b]**

is not equivalent to

B3 = **(a;c;stop |[a]| a;stop)**

2.3.5 The Functionality of LOTOS Behaviour Expressions

The product of the domains of the values passed at successful termination is called the functionality of that termination. For example the functionality of

g?X:Nat Y?Bool; exit(X,Y)

is *domain(Nat) x domain(Bool).*

The definition of any process in LOTOS should indicate its functionality.

process <proc-name><formal-gates><formalparameters>**:**<functionality>**:=**
<behaviour expression>
endproc

The definition of functionality for processes that do not have a sequential structure is indicated by one of the following:

- **noexit** if the process does not terminate successfully at all.
- **exit** if the process terminates successfully without value passing.
- **exit($\mathbf{t_1}$,$\mathbf{t_2}$,...,$\mathbf{t_n}$)** if the process terminates successfully with the values passed $\mathbf{v_1}$,$\mathbf{v_2}$,...,$\mathbf{v_n}$ and $\mathbf{v_i}$ $\in$ domain $\mathbf{t_i}$.

In LOTOS, the functionality of a process is specified in the header after a colon.

EXAMPLE1
process in-out[in,out] : **noexit** :=
in?X:Nat; out!X; **stop**
endproc
EXAMPLE2
process in-out[in,out] : **exit** :=
in?X:Nat; out!X; **exit**
endproc
EXAMPLE3
process in-out[in] : **exit(Nat)** :=
in?X:Nat; **exit(X)**
endproc
The functionality of a behaviour expression **B1 op B2**, where **op** is a LOTOS operator **[]** or **[>** is:
- if **B1** and **B2** terminate successfully, then *func*(**B1 op B2**) = *func*(**B1**) = *func*(**B2**).
- if *func*(**B1**) = **noexit**, the *func*(**B1 op B2**) = *func*(**B2**).
- if *func*(**B2**) = **noexit**, the *func*(**B1 op B2**) = *func*(**B1**).

EXAMPLE
g1?X:Nat;
g2?Y:Nat;
exit(X,Y)
[]
g2?X:Nat;
g1?Y:Nat;
exit(X,Y)
has functionality **exit**(Nat,Nat) which is *domain*(Nat) x *domain*(Nat).
The *parallel composition* of two processes **B1** and **B2** terminates successfully only if **B1** and **B1** terminate successfully with comparable list of values, because **B1** and **B2** must synchronize on the termination value. The functionality of a *parallel composition* behaviour expression **B1 op B2**, where **op** is a LOTOS operator ||,|||, or $\mathbf{[g_1,..g_n]}$ is:

- if **B1** and **B2** terminate successfully, then *func*(**B1 op B2**) = *func*(**B1**) = *func*(**B2**).
- if *func*(**B1**) = **noexit** or *func*(**B2**) = **noexit**, then *func*(**B1 op B2**) = **noexit**.
- if *func*(**B2**) = **noexit**, the *func*(**B1 op B2**) = *func*(**B1**).

EXAMPLE
g1?X:Nat;
g2?Y:Nat;
exit(X,Y)
|||
g3?X:Nat;
stop
has functionality **noexit**.
EXAMPLE
exit(3) ||| **exit**(4)
has functionality **exit**(Nat) which is *domain*(Nat), but the termination will not be successful. This is because both processes will attempt to interact on gate δ with different parameters of the exit.

2.4 Inference Rule Semantics

The operational semantics of LOTOS is defined in terms of inference axioms and rules

[ISO1]. The transitions that a given behaviour expression may perform, and the dynamic behaviour of the next state, can be derived systematically by applying the inference rules.

The *inference axioms* are statements that are assumed to be valid. Examples of axioms are:

$3 > 0$.

$X = X$.

cats are animals.

The *inference rules* are applied on inference axioms and other inference rules to derive new valid statements. Inference rules have the form:

If S_1 and S_2 ... and S_n are valid then S_m is also valid

Examples of inference rules are:

If $X > 3$ and $X < 5$ then $X = 4$.

If X is a cat then X is an animal.

Inference rules are often written in the form:

$$\frac{S_1,\ldots,S_n}{S_m}$$

Then the above example can be written as:

$$\frac{X > 3,\ X < 5}{X = 4}$$

The notation B1 -**a**-> B2 denotes a *labelled transition*, and means that B1 may perform the transition **a** and the resultant behaviour will be B2. For example:

$$\frac{B1 \text{ -}\mathbf{a}\text{-> } B3}{B1 \text{ [] } B2 \text{ -}\mathbf{a}\text{-> } B3}$$

is an inference rule which states that behaviour B1 [] B2 can perform action **a** resulting in behaviour B3, if B1 can performaction **a** resulting in B3.

In the rest of this section, we present the inference rules that define the operational semantics of LOTOS. These rules are essentially the same as those presented in [ISO1].

In order to define the inference rules for LOTOS behaviour expressions, let:

eval(**E**) denote the value of the term **E**;

G denote the set of the behaviour's formal gates;

$\mathbf{g}, \mathbf{g_i} \in G$;

i denote an internal action;

$\mathbf{d_i}$ is either

$!\mathbf{E_i}$, denoting the offering of the value eval($\mathbf{E_i}$)

or

$?x_i:s_i$, denoting that a value for the variable $\mathbf{x_i}$ of sort $\mathbf{s_i}$ is expected;

d denote the successful termination's action name;

a denote any action

name(a) gives the gate identifier of action **a**

(**name(a)** $\in$ **G** U **{i, d }**);

$\mathbf{t_i}$ is a term (ADT value expression);

[t_i] or **eval(t_i)** is the ADT evaluation of **t_i**;

[t_1/x_1,...,t_n/x_n] B denotes the result of the replacement of all occurrences of **x_1,..,x_n** in **B** by **t_1,..,t_n** respectively.

(B)[g_i/g] denote that **g** is relabelled by **g_i** for every action that may be performed by **B** on the gate **g**.

2.4.1 Inference Axioms

In LOTOS semantics, axioms describe actions that can be derived directly from the given behaviour.

The following rules generate the axioms:

2.4.1.1 Action Prefix

B = **a;B'**

JJ

where B is ready to offer the action **a** resulting in behaviour **B'**.

The action **a** can be an internal action **i** which can always be executed, or an observable action which may be executed if the environment can synchronize with it.

J

J

a) Internal Action Prefix

i;B **-i->** B is an axiom

EXAMPLE

i;a;**stop** **-i->** a;**stop** is a transition

b) Observable Action Prefix

$gd_1..d_n$;B -$gv_1..v_n$-> [ty_1/y_1,..,ty_m/y_m]B is an axiom

iff

v_i = eval(t_i) for d_i = !t_i ($1 \le i \le n$),

v_i is a value of sort s_i for d_i = ?x_i:s_i ($1 \le i \le n$),

ty_1,..,ty_m are term instances with v_i = eval(ty_i) if

d_i = ?y_j:s_j ($1 \le i \le n$, $1 \le j \le m$, $m \le n$) and

{y_1,..,y_m} = {x_i| d_i = ?x_i:s_i, $1 \le i \le n$}.

The example below clarifies the above definitions.

EXAMPLE

g1?X:Nat!0; g2!Succ(X);**stop**

-g1 eval(T) eval(0)-> g2!Succ(T);**stop**

is a transition iff eval(T) $\in$ domain(Nat)

By the above definitions we have:

n = 2

m = 1

d_1 = ?X:Nat

d_2 = !0

y_1 = X

ty_1 = T

v_1 = eval(T) $\in$ domain(Nat)

v_2 = eval(0) = 0

The intuitive meaning of the notation - **g eval(T)** -> is that the environment provided value expression **T** at gate **g**.

c) Observable Action Prefix with Selected Predicates

$gd_1..d_n[P];B$ -$gv_1..v_n$-> B' is an axiom

iff

$gd_1..d_n;B$ -$gv_1..v_n$-> B' is an axiom

and

$eval([ty_1/y_1,..,ty_m/y_m]P)$ = true,

where

[P] is a predicate,

$ty_1,..,ty_m$ are term instances with $v_i = eval(ty_i)$ if

$d_i = ?y_j:s_j$ ($1 \leq i \leq n$, $1 \leq j \leq m$, $m \leq n$) and

$\{y_1,..,y_m\} = \{x_i | d_i = ?x_i:s_i, 1 \leq i \leq n\}$.

EXAMPLE

g?X:Nat[X>Succ(0)];**stop** -g eval(T)-> **stop** is a transition iff

eval(T) $\in$ domain(Nat), and

eval(eval(T) > Succ(0)) = **true**

2.4.1.2 Successful Termination

a) Without Value Passing

exit - δ -> **stop** is an axiom.

b) With value Passing

exit($E_1,..,E_n$) - δ $v_1..v_n$-> **stop** is an axiom iff

$v_i = eval(E_i)$ if E_i is a term ($1 \leq i \leq n$)

$v_i \in domain(s_i)$ if E_i = **any** s_i ($1 \leq i \leq n$).

EXAMPLES

exit - δ -> **stop**

exit(largest(Succ(0),0), **any** Bool) - δ Succ(0) V->**stop**

iff V $\in$ domain(Bool) (e.g. V equals **true** or **false**)

are transitions

2.4.2 Inference rules

Inference rules are needed to derive all possible transitions that can be fired by behaviour expression constructs other than action prefix.

2.4.2.1 Local Definition

$[t_1/x_1,..,t_n/x_n]$ B' -a-> B"

---

let $x_1:s_1=t_1,..,x_n:s_n=t_n$ **in** B' -a-> B"

is an inference rule.

EXAMPLE

let B =

let X:Nat=Succ(0), Y:Bool=true **in**

g1!X; g2!Y; **stop**

then
B -g1 Succ(0)-> g2!true; **stop** is a transition.

2.4.2.2 Guard

B’ -a-> B"

[P] -> B’ -a-> B"

where [P] is a guard, is an inference rule if eval(P) = true.

EXAMPLE

let B =

[Succ(0) > 0] ->

g!largest(Succ(0),Succ(Succ(0)));**stop**

then

B -g Succ(Succ(0))-> **stop** is a transition.

If, for example, the guard in B is [0 > 0] then B is equivalent to **stop**.

2.4.2.3 Choice

B_1 -a-> B_1’

B_1 [] B_2 -a-> B_1’

B_2 -a-> B_2’

B_1 [] B_2 -a-> B_2’

are inference rules.

EXAMPLE

let B =

a;b;**stop**

[]

c;d;**stop**

then

B -a->b;**stop**

and

B -c->d;**stop**

are the valid transitions.

2.4.2.4 Hiding

B’ -a-> B"

---

hide g_1,..,g_n **in** B’ -a-> **hide** g_1,..,g_n **in** B"

is an inference rule if name(a) $\notin \{g_1,..,g_n\}$

B’ -a-> B"

---

hide g_1,..,g_n **in** B’ -**i**-> **hide** g_1,..,g_n **in** B"

is an inference rule if name(a) $\in \{g_1,..,g_n\}$

EXAMPLE

let B =

hide g1,g2 **in**
g1; g2; **stop**
[]
g3; g1; **stop**
then
B **-i**-> **hide** g1,g2 **in** g2;**stop**
and
B -g3-> **hide** g1,g2 **in** g1;**stop**
are the valid transitions.

2.4.2.5 Nested

B -a-> B'

(B) -a-> B'
is an inference rule.
EXAMPLE
(a;b;**stop**) -a-> b;**stop**
is a transition.

2.4.2.6 Parallel

a) selected synchronization

B_1 -a-> B_1', B_2 -a-> B_2'

B_1 |[g_1,...,g_n]| B_2 -a-> B_1' |[g_1,...,g_n]| B_2'
if name(a) $\in$ {g_1,..,g_n, δ }

This case describes the situation where the two behaviours interact.

B_1 -a-> B_1'

B_1 |[g_1,...,g_n]| B_2 -a-> B_1' |[g_1,...,g_n]| B_2
if name(a) $\notin$ {g_1,..,g_n, δ }

B_2 -a-> B_2'

B_1 |[g_1,...,g_n]| B_2 -a-> B_1 |[g_1,...,g_n]| B_2'
if name(a) $\notin$ {g_1,..,g_n, δ }

are inference rules.
These two cases describe the situation where no interactions occur. In these cases the behaviours proceed independently.
EXAMPLE
let B =
a;b;**stop** [] c;a;**stop**
|[a]|
d;e;**stop** [] a;d;**stop**
then
B -a-> b;**stop** |[a]| d;**stop**
B -c-> a;**stop** |[a]| d;e;**stop** [] a;d;**stop**

B -d-> a;b;**stop** [] c;a;**stop** |[a]| e;**stop**

are valid transitions.

b) interleaving

B_1 |[]| B_2 -a-> B'

B_1 ||| B_2 -a-> B'

is an inference rule.

EXAMPLE

let B =

exit(**any** Nat) [] a;d;**stop**

|||

a;e;**stop** [] exit(Succ(0))

then

B - δ Succ(0)-> **stop** ||| **stop**

B -a-> d;**stop** ||| a;e;**stop** [] exit(Succ(0))

B -a-> exit(**any** Nat) [] a;d;**stop** ||| e;**stop**

are valid transitions.

c) full synchronization

B_1|[g_1,..,g_n]|B_2 -a-> B'

B_1 || B_2 -a-> B'

is an inference rule where {g_1,..,g_n} = G, is the union of all possible gates of B1 and B2.

EXAMPLE

let B =

exit(**any** Nat) [] a;d;**stop**

||

a;e;**stop** [] exit(Succ(0))

then

B - δ Succ(0)-> **stop** || **stop**

B -a-> d;**stop** || e;**stop**

2.4.2.7 Enable

B_1 -a-> B_1'

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

if name(a) ≠ δ .

This is the case where the enabling behaviour B1 continues.

B_1 - δ v_1,..,v_n-> B_1'

B_1 >> **accept** x_1:s_1,..,x_n:s_n **in** B_2 **-i**-> [t1/x1,..,tn/xn]B2

are inference rules where

$v_i = eval(t_i)$ $(1 \leq i \leq n)$.

This is the case where the enablling behaviour B1 terminates successfully by the transition at gate δ (see **exit** axioms). The values of that transition are passed to B2 by an internal action **i**, then B2 takes over.

EXAMPLE 1

let B =

a;b;**exit** [] **exit** >> c;d;**stop**

then

B -a-> b;**exit** >> c;d;**stop**

B -**i**-> c;d;**stop**

are valid transitions.

Note that the **accept** statement is optional. It is needed only if value passing is required.

EXAMPLE 2

let B =

a?X:Nat;b!X;**exit**(X, X > Succ(0)) [] **exit**(**any** Nat, **any** Bool)

>> **accept** Y:Nat, Z:Bool **in**

c!Y;d!Z;**stop**

then

B -a eval(T)-> b!T;**exit**(T, T > Succ(0))

>> **accept** Y:Nat, Z:Bool **in**

c!Y; d!Z; **stop**

B -**i**-> c!N; d!B; **stop**

where eval(T),eval(N) $\in$ domain(Nat) and

eval(B) $\in$ domain(Bool)

are valid transitions.

2.4.2.8 Disable

B_1 -a-> B_1'

---------- if name(a) $\neq \delta$

B_1 [> B_2 -a-> B_1' [> B_2

B_1 - δ $v_1..v_n$-> B_1'

B_1 [> B_2 - δ $v_1..v_n$-> B_1'

B_2 -a-> B_2'

B_1 [> B_2 -a-> B_1'

are inference rules

EXAMPLE

let B=

a;b;**stop** [] **exit**

[>

c;d;**stop**

then

B -a-> b;**stop** [> c;d;**stop**

B - δ -> **stop**
B -c-> d;**stop**
are valid transitions.

2.4.2.9 Summation on Values

[t/x] B' -a-> B"
---
choice x:s [] B' -a-> B"
is an inference rule iff eval(t) ∈ domain(s).
EXAMPLE
let B =
choice x:Nat []
a!x[x mod 3 = 0];**stop**
then
B - a 3 -> **stop**
B - a 6 -> **stop**
B - a 9 -> **stop**
...
...
are valid transitions

2.4.2.10 Relabelling

We recall that the relabelling behaviour expression appears only in the dynamic semantics of the behaviour expressions, and it is not in the syntax of LOTOS.

B' -g $v_1..v_n$-> B"

(B')[$g_1/h_1,..,g_n/h_n$] -g' $v_1..v_n$-> (B")[$g_1/h_1,..,g_n/h_n$]
with
g' = g if g ∉ {$h_1,..,h_n$}
g' = g_i if g = hi ($1 \leq i \leq n$),
is an inference rule.
EXAMPLE
let B =
(a;b;**stop** [] c;a;**stop**)[d/a,e/b]
then
B -d-> (b;**stop**)[d/a,e/b]
B -c-> (a;**stop**)[d/a,e/b]
are valid transitions.

2.4.2.11 Summation on Gates

(B')[g_i/g] -a-> B"
---
choice g **in** [$g_1,..,g_n$] [] B' -a-> B"
is an inference rule for each $g_i \in \{g_1,...,g_n\}$.
EXAMPLE
let B=

choice a **in** [g_1,g_2] []

a;b;a;**stop**

[]

c;d;**stop**

then

B -g_1->(b;a;**stop**)[g_1/a]

B -c->(d;**stop**)[g1/a]

B -g_2->(b;a;**stop**)[g_2/a]

B -c->(d;**stop**)[g_2/a]

are valid transitions.

2.4.2.12 par-Expression

B'[g_1/g] op ... op B'[g_n/g] -a-> B"

---

par g **in** [g_1,...,g_n] op B' -a-> B"

is an inference rule where

g,g_1,..,g_n are gate-variable instances,

op is a parallel-operator.

EXAMPLE

let B =

par a **in** [g1,g2] |||

a;b;a;**stop**

then

B -g1-> (b;a;**stop**)[g1/a] ||| (a;b;c;**stop**)[g2/a]

B -g2-> (a;b;a;**stop**)[g1/a] ||| (b;a;**stop**)[g2/a]

are valid transitions.

2.4.2.13 Process Instantiation

([t_1/x_1,...,t_m/x_m]B)[g_1/h_1,..,g_n/h_n] -a-> B'

---

p[g_1,..,g_n](t_1,..,t_m) -a-> B'

is an inference rule

iff

there exist a process definition:

p[h_1,..,h_n](x_1:s_1,..,x_m:s_m) := B

EXAMPLE

suppose we have the following process definition:

process any-thing[g1,g2](X:Nat;Y:Bool) : **noexit** :=

g1!X; g2!Y; **stop**

[]

g2!Y; g1!X; **stop**

endproc

and let B =

any-thing[a,b](largest(Succ(0),0), (0 > Succ(0)))

then

B -a Succ(0)-> (g2!(0 > Succ(0));**stop**)[a/g1,b/g2]

B -b false-> (g1!largest(Succ(0),0);**stop**)[a/g1,b/g2]

are valid transitions.

2.4.2.14 stop

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

2.4.3 Example

<

Here, we demonstrate how the inference rules can be applied to obtain the possible transitions of a given behaviour expression.

Suppose the following process definition exists:

process P[a,b]:**noexit**:=

a;b;**stop**

[]

b;a;**stop**

endproc

and Let B=

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$

1-> Find all

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ -trans-> B'

By applying the inference rules of selected synchronization we obtain:

$P[g_1,g_2]$ -trans-> B_1, $P[g_2,g_3]$ -trans-> B_2

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ -trans-> B_1 $|[g_2]|$ B_2

if name(trans) $\in \{g_2, \delta\}$

and

$P[g_1,g_2]$ -trans-> B_1

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ -trans-> B_1 $|[g_2]|$ $P[g_2,g_3]$

if name(trans) $\notin \{g_2, \delta\}$

and

$P[g_2,g_3]$ -trans-> B_2

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ -trans-> $P[g_1,g_2]$ $|[g_2]|$ B_2

if name(trans) $\notin \{g2, \delta\}$

2-> To satisfy 1, we have to find all

P[g1,g2] -trans1-> B1

and P[g2,g3] -trans2-> B2

We have:

(a;b;**stop []** b;a;**stop**) $[g_1/a,g_2/b]$ -trans-> B_1

a) ---

$P[g_1,g_2]$ -trans-> B_1

and
(a;b;**stop []** b;a;**stop**) [g_2/a,g_3/b] -trans-> B_2

b) ----------

P[g_2,g_3] -trans-> B_2

3-> To satisfy 2(a) we have to find all

(a;b;**stop []** b;a;**stop**) [g_1/a,g_2/b] -trans-> B_1

We have:

a;b;**stop []** b;a;**stop** -g $v_1..v_n$-> B_1'

(a;b;**stop []** b;a;**stop**)[g_1/a,g_2/b] -g' $v_1..v_n$-> (B_1')[g_1/a,g_2/b]

g' = g if g $\notin$ {g_1,g_2}

g' = g_1 if g = a

g' = g_2 if g = b

4-> Find all

a;b;**stop []** b;a;**stop** -trans-> B_1'

We have:

a;b;**stop** -trans-> B_1'

---

a;b;**stop []** b;a;**stop** -trans-> B_1'

and

b;a;**stop** -trans-> B_1'

---

a;b;**stop []** b;a;**stop** -trans-> B_1'

5-> By the axiom of prefix behaviour we can obtain directly the following transitions:

b;a;**stop** -b-> a;**stop**

a;b;**stop** -a-> b;**stop**

4<- Back to step 4, we now can obtain the following transitions:

a;b;**stop []** b;a;**stop** -a-> b;**stop**

a;b;**stop []** b;a;**stop** -b-> a;**stop**

3<- Back to step 3, the following transitions can then be obtained:

(a;b;**stop []** b;a;**stop**) [g_1/a,g_2/b] -g_1-> (b;**stop**)[g_1/a,g_2/b]

(a;b;**stop []** b;a;**stop**) [g_1/a,g_2/b] -g_2-> (a;**stop**)[g_1/a,g_2/b]

2<- In step 2(a) above we have:

P[g_1,g_2] -g_1-> (b;**stop**)[g_1/a,g_2/b]

P[g_1,g_2] -g_2-> (a;**stop**)[g_1/a,g_2/b]

are valid transitions.

Similarly for step 2(b) we can obtain:

P[g_2,g_3] -g_2-> (b;**stop**)[g_2/a,g_3/b]

P[g_2,g_3] -g_3-> (a;**stop**)[g_2/a,g_3/b]

1<- Then from the initial behaviour expression we can obtain the following transitions:

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ $-g_2->$

(a;**stop**)$[g_1/a,g_2/b]$

$|[g2]|$

(b;**stop**)$[g_2/a,g_3/b]$

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ $-g_1->$

(b;**stop**)$[g_1/a,g_2/b]$

$|[g_2]|$

$P[g_2,g_3]$

and

$P[g_1,g_2]$ $|[g_2]|$ $P[g_2,g_3]$ $-g_3->$

P[g1,g2]

$|[g_2]|$

(a;**stop**)$[g_2/a,g_3/b]$

Chapter 3: Functionality of ISLA

3.1 Introduction

The dynamic behaviour of a LOTOS specification can be seen as a tree, called Behaviour Tree, where the nodes of the tree represent the states of the behaviour, and the arcs represent the possible "next actions". For example for the following behaviour in LOTOS:

i;stop
[]
a;(b;stop [] c;stop)

the corresponding Behaviour Tree is :

(bh0)
/ \
i / \ **a**
/ \
(bh1) (bh2)
/ \
b / \ **c**
/ \
(bh21) (bh22)

where
bh0 = original behaviour
bh2 = **(b;stop [] c;stop)**
bh1,bh21 and bh22 = **stop**

The purpose of ISLA is to prototype the dynamic behaviour of LOTOS specifications by allowing the user to traverse the behaviour tree of a given specification, and check whether or not the exercised branches (sequences of actions) conform to the intented behaviour. The basic operation of ISLA is as follows. From the current behaviour expression, a list of all acceptable actions is given. Subsequently, the user chooses one of the actions, identified by a unique number, and if this action involves a choice of data, the user is required to provide it. The dynamic behaviour of the next state is then derived. This operation can be repeated under user guidance.

In addition, ISLA supports other functions allowing users to explore several alternative execution paths in the Behaviour Tree. It is possible to backtrack the execution to any point on the executed path from the root to the current state. "Checkpoints" can also be set at any visited node. Therefore, as an alternative for moving down and up the tree, it is possible to jump directly to a specific previously set checkpoint. A trace of all explored paths can be displayed.

A checkpoint can be saved on an external file, giving the user the possibility of restarting the execution some time later. Any analysis report generated by ISLA can be saved on external files.

ISLA also allows symbolic execution of any process behaviour, generating the tree of all possible next actions, with some limitations described later. Help can be provided when dealing with complex value expressions and guards.

At any point during simulation, the user can obtain a list of all available commands. The ability of obtaining the description of a specific command is also provided.

During simulation, if a value expression needs to be validated or evaluated, the ADT interpreter **SVELDA** is called. These value expressions could be included explicitly in the specification, or entered by the user in the case where a choice of data of a specific sort is required.

ISLA has been designed to have the following features:
- enough analysis functions to cover all important aspects of the specification under simulation.
- an appropriate user interface, where help can be obtained at any point of the simulation.
- acceptable costs in terms of CPU and memory usage.
- adaptability to changes.

The simulation is divided into three phases:
I- *Starting phase:* upon running ISLA, a menu (called main menu) of all possible initial commands is displayed. This offers the possibility of:
- listing the existing source and 'internal form' LOTOS specifications.
- listing the checkpoints files.
- compiling a source LOTOS specification (done by a system call to the LOTOS compiler).
- simulating an 'internal form' LOTOS specification or resuming simulation at an existing checkpoint and
- executing any UNIX command.

II- *Process Menu phase:* in this phase a menu of all processes of the chosen LOTOS specification is displayed. The simulation can be done for any chosen process.
III- *Simulation phase:* a menu of all possible actions is displayed.

We refer users to Appendix B where a LOTOS example is constructed. Furthermore, in Appendix C, we show how ISLA can be used to simulate it. In this way, we illustrate the most important functionalities of the interpreter.
In this chapter, a more complete description of the functions supported by ISLA is given (Appendix D includes commands help in different phases).
In the remainder of this thesis, we use different terms that have the same meaning, such as:
- 'state', 'point', 'node', 'behaviour expression', 'dynamic behaviour', or simply 'behaviour'.
- 'next state', 'subsequent node', or 'resulting behaviour' after executing an action.
- 'interaction', 'branch', 'event', or 'action'.

3.2 Function Description

The major functionalities of ISLA are described in the following sections.

3.2.1 Menu of existing Processes

When an internal form LOTOS specification is loaded for execution, a menu of its process headers is listed (phase II), where each header includes process name, formal gates, formal parameters and a unique number (starting with the number of the main specification, which is usually '1'). The following is the process menu of the pop machine example described in Appendix B, where the specification name is 'system', where 'system' contains three subprocesses 'pop-machine', 'get-money', and 'deliver'.

==========

The available processes are:
[1] system[coin-in,buttons,drawer]()
[2] pop-machine[coin-in,buttons,drawer]()
[3] get-money[coin-in,buttons](amount:COIN)
[4] deliver[buttons,drawer]()
P:Choose one command or 'lp' or 'h' for help ==>

==========

Example: menu of available processes (Phase II)

To execute (simulate) a process, its number should be entered, followed by a list of actual gates (optional) and actual value parameters.

If the input is found to be free of syntax and static semantics errors, then the execution goes from phase II to phase III (simulation phase).

3.2.2 Menu of Possible Actions and Next Behaviours

At each step during simulation (phase III), a menu of all possible actions and their resulting behaviours is given. In terms of the Behaviour Tree, the menu includes all the branches of the current node and the subsequent nodes. Since each subsequent node represents a behaviour expression which may be very large, unique indexed symbols are used to represent nodes in the menu.

An action in the menu has the following form:

<*i*> - *action* ----> **bh***i* **[***ln*$_1$,..,*ln*$_m$**]**

It can be read as: "The current behaviour expression can accept the action '*action*', and will result in behaviour **bh***i*".

where:

i is the number associated with the given action (index)

bh*i* 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$_1$,.. and action$_m$) in the source external specification. *action* can be :

i representing an internal event

<gate-name> <$d_1 ... d_n$> [*P*] (n,m > 0) where

d_i = **!**eval(**t$_i$**) where **t$_i$** the offered term, or

= **?x$_i$:s$_i$** or

= **?[x$_1$,..,x$_m$]:s$_i$** which indicates an interaction event for **?x$_1$:s$_i$**, **?x$_2$:s$_i$** ... and **?x$_m$:s$_i$** .

[P] is the union of selection predicates of all interacting actions.

The following is an example of an action menu taken from the simulation of the pop machine specification.

========== **system** ==========

CP/MANUAL W/20 Level/2 Path/[1,1]

Events [1] coin-in ?$50:COIN
[1,1] coin-in ?$25:COIN
==========
<1>- coin-in ?coin:COIN ----> bh1 [51]
<2>- buttons?[button-name,button-name]:BUTTON
[ge(0+$50+$25,price(button-name))] ----> bh2 [53,57]
==========

Example: menu of possible actions (Phase III)

Action # 2 is an interaction between processes 'get-money' and 'deliver' at gate 'buttons'. The two participating processes must agree with the environment on a value of type 'BUTTON' for the two variables 'button-name', taken from process 'get-money', and 'button-name', taken from process 'deliver', with the restriction that price(button-name) should be less that or equal to 75 cents. The two actions involved in this interaction are at line 53 and 57 in the external source.

The menu also shows:

- The Execution Level : this is the number of actions fired starting from the initial state and leading to the current state, and is equal to the length of the path from the initial state to the current state in the **BT** (Behaviour Tree) of the current process.
- The Execution Path : this is the list of the numbers of the actions fired starting from the initial state and leading to the current state. (Note that the Execution Level is the length of the Execution Path).
- The Check Points Setting Status, and the maximum width of the execution tree (described later).
- The events that were taken on the path from the initial to the current state (optional).

An action can be chosen by giving its associated number, and if it requires values from the environment, the user has to provide them. The action is rejected if its associated predicates (if any) are evaluated to 'false' because of the values involved. In this case the user is notified, and the execution remains in the same state.

The *pseudo-action* of the **choice** construct is represented as:

***<*i*> - dummy?x_j:s_j ----> bh*i* [*ln*]**

which means that the action that can be performed by $\mathbf{bh}_i$ can be determined only if a value for $\mathbf{x_j}$ of type $\mathbf{s_j}$ is entered. The reason of this representation is described in chapter 4.

3.2.3 The Back and Level functions

Moving down the Behaviour Tree of a specification will cover one sequence of interactions. ISLA provides the user with the ability to explore additional sequences by allowing the execution to back up by any number of steps. The user then can choose another action at this state and move to a different subsequent node.

Alternatively, if this is more convenient, the user can restart the simulation at a specific level. This is shown in the example of Appendix C (Part 8).

3.2.4 Check Point Setting

During exploration of the Behaviour Tree, it is possible to maintain the behaviour expression of any visited node in memory, thereby giving the user the possibility to jump directly

to any of these nodes anytime during simulation. These nodes are called "checkpoints". See Appendix C (Part 6).

One of the important features of ISLA, is that it allows to save a "checkpoint" into an executable external file, so that the user can resume the simulation at that point any time later. This is helpful when the user cannot finish his/her intended analysis in one session.

The setting of "checkpoints" can be done *manually*, where the user chooses which visited node is to be set, or *automatically*, where ISLA sets a "checkpoint" at every visited node (e.g. useful for exhaustive coverage). The latter requires a large amount of memory for large specifications. A "checkpoint" includes the current behaviour expression, the sequence of actions that led to this point, with their associated numbers (execution path), and the number which is the name of the checkpoint. Checkpoints are numbered consecutively.

3.2.5 Menu of Check Points

During simulation (phase II), a menu of all existing "checkpoints" can be listed.

```
============================================================
The available Check Points are:
-<1>- [1,2]
-<2>- [2,2,1]
-<3>- [2,1,2,2,1]
============================================================
A:Choose one command or 'la' or 'h' for help ==>
============================================================
```

Example: menu of available Check Points

Each "checkpoint" is identified by its associated number (the order of its setting), and the execution path from the root of the Behaviour Tree to that point. We recall that an execution path is an action sequence, represented by the numbers of the actions in that sequence. For example, the execution path [2,2,1] means that action number 2 was taken at level 0, subsequently, action number 2 was taken at level 1, then action number 1 at level 2. It is possible to reexecute the same sequence with different choices of values. In this case we will have different sequences of actions with the same execution path representation.

At any point, the user can:

- jump to a specific "checkpoint"
- remove one or all existing "checkpoints"
- see the trace of actions that were taken to lead to a specific "checkpoint".
- see the External representation of a Check Point
- save a Check Point into an Executable file for later execution.
- obtain the Behaviour Tree of a Check Point using symbolic execution.

A detailed description of the above functions follows in the next sections.

3.2.6 Saving a Check Point into an Executable File

As a further help for the user, any state of execution (the current state or any "checkpoint" in the 'Check Point' menu), can be saved into an executable file, giving the user the possibility to resume execution some time later.

A checkpoint can be represented in the external file by the sequence of actions that led to it, so that, when the execution is required to resume at that checkpoint later on, an automatic execution will be done using the information given in the sequence of actions to obtain the

behaviour expression of the intended point. To avoid this automatic execution, the user can choose to save the behaviour expression with the sequence of actions.

3.2.7 User-Defined Data

ISLA allows users to define shorthands for commonly used constants, as well as sets of constants that can be used when a choice of values is required to satisfy a predicates.

3.2.7.1 Constants

Constants for LOTOS value expressions can be defined by giving each constant a unique shorthand name preceded by the character '$'. A constant can be defined in terms of other already defined constants. The following example shows how constants to represent Natural numbers 0 through 9 can be defined.

$0 = 0
$1 = Succ($0)
$2 = Succ($1)
$3 = Succ($2)
$4 = Succ($3)
$5 = Succ($4)
$6 = Succ($5)
$7 = Succ($6)
$8 = Succ($7)
$9 = Succ($8)

Example: Constant definition

At the time of definition, constants are syntactically and semantically checked, translated into an internal form, and added with their corresponding sorts to the Constant Data Base. The duplication of shorthand names is also checked. These constants can be saved into an external file, thereby giving the user the possibility of using the same constants for another session. Another example is shown in Appendix C (part 5).

Constant definitions can be saved in their external form representation. Alternatively, their internal form representation can be saved. Constants saved in an internal form, however, can only be used for the specification that was under simulation when these constants were saved, because the internal form representation includes the renaming of operators and sorts, which will differ between one specification and another.

These constants then can be chosen by giving their shorthand name. They can also be a part of any other value expressions, for example if the constants in the previous example were defined and the user wanted to enter the Natural number 11 as a choice of value for a certain event, then (s)he can simply enter 'Succ(Succ($9))'.

3.2.7.2 Value Expression Sets

ISLA also allows to define sets of value expressions, where each set contains value expressions of one particular sort, and sets can be identified by giving them unique numbers. Again each value expression of a set is syntactically and semantically checked and translated into internal form.

Succ($9)
Succ(Succ(0))
largest($4,$1)
$1

Example: Set 1 of sort Nat

true

not(true)

false

Example: Set 2 of sort Bool

When complex selection predicates are involved in an action, it might be difficult to find values that satisfy them. In this case the user can assign for each event that requires a value in that action a particular set. Then the predicates are evaluated for each possible combination of values taken from the sets. ISLA then reports the evaluation result for each tuple of values, and a unique number is given to each evaluation. The user then can choose any tuple that caused the predicates to be successfully evaluated to be the next choice of values. It is also possible to obtain detailed step-by-step traces of an evaluation process that caused the predicates to fail.

If for example we have the following action that was offered in the menu of "next actions" :

g?X:Nat ?Y:Bool ?Z:Nat [X > Z and Y = true]

and the user assigns set 1 of example ... to the event **?X:Nat**, set 2 to the event **?Y:Bool**, and the value expression **$2** for the event **?Z:Nat**, then the following menu will appear:

```
============================================================
Guard is : [X > Z and Y = true]
        X           Y       Z           result
------------------------------------------
1> Succ($9)true   $2      true
2> Succ($9)not(true)$2false
3> Succ($9)false$2    false
4> Succ(Succ(0))true$2false
5> Succ(Succ(0))not(true)$2false
6> Succ(Succ(0))false$2false
7> largest($4,$1)true$2true
8> largest($4,$1)not(true)$2false
9> largest($4,$1)false$2false
10> $1          true    $2      false
11> $1          not(true)$2false
12> $1          false$2     false
============================================================
Command Line =>
============================================================
```

Example: Menu of an evaluated predicate

The usefulness of this feature of course is especially evident in the case of more complex selection predicates.

Appendix C (Part 9 & 10) shows another example.

3.2.8 External Displays

During simulation, ISLA can provide users with the external representation of any process definition or any behaviour expression. The output will be justified corresponding to the priority levels of the operators, as in the example below.

```
process p[g1,g2]:=
    a;
b;
stop
```

[]
c;
stop
[>
d;
stop
endproc

See Appendix C (Part 11) for different example.

3.2.9 Valid Sort Equations

When value expressions of specific sorts are required, such as actual process parameter values or interaction values, the user is able to list the valid syntax of such values. See Appendix C (Part 4).

For example the output of value expressions syntax for the formal parameters of

P[g1,g2](X:*Nat*,Y:*Bool*,Z:*Nat*)

where the sorts Nat and Bool are defined in the standard library, will be of the following format:

Nat -->
0
Succ(*Nat*)
largest(*Nat*,*Nat*)

Bool -->
true
false
not(*Bool*)

Nat = *Nat*
Nat > *Nat*
Nat < *Nat*
...
...

Therefore the actual parameters list:

(largest(Succ(0),0)), not(true), 0)

is valid.

When the syntax for the value expression of a given sort contains another sort, then the syntax of the value expressions of the latter sort will also be listed.

3.2.10 Menu of evaluated Guards and Value Expressions

During the process of applying the inference rules to obtain the possible "next actions" for the current behaviour expression, ISLA saves all the guards and value expressions that were executed. A menu of all these expressions and their evaluations can be listed. The user is allowed to see a trace of a certain evaluation and to ’force’ certain unsuccessful guards evaluations to be successful. Then the inference rules can be reexecuted on the same behaviour, and this may cause some extra actions to be offered. This is helpful when a guard was intended to be successful, but, due to an improper ADT equations or due to a certain value that was entered earlier during the simulation, the guard was unsuccessfully evaluated.

3.2.11 Analysis Reports

Other than the above functions, ISLA can generate reports to help the user for better analysis. These reports can be printed on the screen or they can be directed into external text files.

3.2.11.1 Symbolic Execution

The "step-by-step" execution, which has been described so far, is inevitably slow, because of the constant interaction with the user. It is possible to produce the Behaviour Tree of a specification (see section 3.1), where all execution paths are shown up to a maximum specified depth, by using "symbolic execution". The tree is computed completely by using backtracking, and replacing actual variable values by symbolic names, according to well-known principles in computer science [BJ]. The predicates that involve variables whose values depend on interactions with the environment are assumed to be 'true' and will be shown in the Symbolic Behaviour Tree. Since such a tree can have a large depth and width (often infinite), the user is expected to provide these upper bounds. An example of a symbolic tree is provided in Appendix C (Part 13).

Depending on the format chosen, the actions in the tree may be presented by their gates only, also the process instantiations between actions can be shown.

The Symbolic Behaviour Tree can be obtained from Phase II for the behaviour of any process, or from Phase III for the current or a resulting behaviour expression, or for a behaviourexpression of an existing Check Point. This subject is developed further in section 4.11.

Research is being carried out on applying the congruence equivalence properties, described in Milner's CCS, on dynamic behaviour expressions to obtain finite symbolic trees [LG].

3.2.11.2 History of Actions

The user may want to keep track of all the actions taken so far in the current session, this will allow him/her to see all the execution paths that were explored. The history trace is represented in a tree form. The information that is given for each action is:

- the action (includes the gate and the events with their chosen value expressions)
- its associated number in the "next actions" menu
- a list of line numbers for where this action occurs in the source specification (it may be more than one line number, in the case of synchronization).
- if a Check Point is set at a specific state in the history tree, then its associated number will be shown
- an indication of the position of the current execution point (illustrated by asterisk).
- if required, the process instantiations between any two sequential actions can be shown.

An example is provided in Appendix C (Part 12).

The following different kinds of history can be provided:

a) Action History

This includes only one path that leads from the initial state (initial call) to the current state. This command can also be used to determine the Point History of a Check Point.

b) Process History

Process history shows all the actions taken so far for the initial process call (e.g. the above example).

c) All History

In one session of simulation, the user can execute many processes, or the same process but with different parameters. ISLA can show the process history of each process call.

Chapter 4: ISLA Design and Implementation

4.1 General Structure

This chapter describes the overall structure of ISLA (Interactive System for LOTOS Applications) and the implementation of important components of its functionalities.

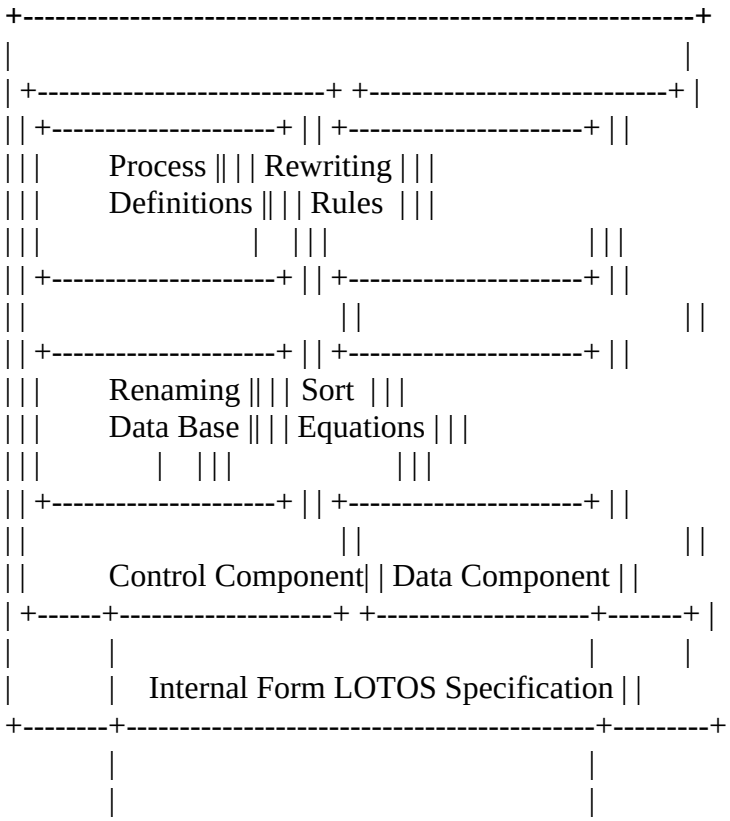
ISLA is programmed in Quintus Prolog under UNIX. Prolog was chosen as the implementation language for the following reasons:

- Objects (e.g. the internal form of LOTOS process definitions, renaming database, axiom rules, etc...) can be described easily using facts.
- Rules can be used to derive other objects (e.g. inference rules, dynamic behaviours, etc...).
- Prolog supports dynamic allocations for facts and rules, unification, direct substitution and backtracking. These are all of great importance in the implementation of ISLA.
- Individual functionalities can be easily added, tested, modified, or removed.

In addition, Prolog is widely used in research environments.

We recall that ISLA is a **Behaviour Interpreter**, and runs on the 'internal' form of the **control** part. ISLA also uses the **ADT Interpreter**, called SVELDA (System for Validating and Executing LOTOS Abstractions). The ADT interpreter runs on the 'internal' form of the **data** component [FEH].

Figure 4.1 shows the general structure of the above interpreters. The relations between the components are also shown.



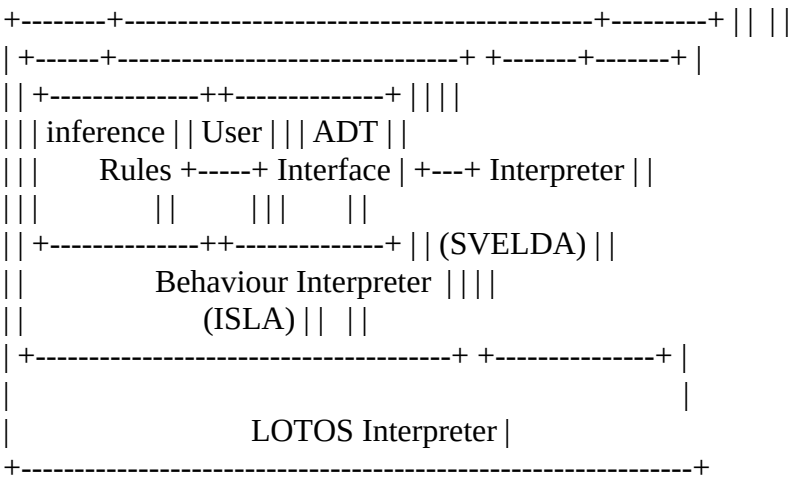


Figure 4.1 Structure of the Interpreter

4.2 Internal Form Representation

The internal representation of a LOTOS specification has the following features:

- It is free of syntactic and semantic errors.
- It is well structured in a way that no scanning is needed to obtain specific information.
- It is equivalent to the source so that the latter can be recovered from the internal representation, with some exceptions such as process functionality, comments, spacing, etc.

The 'internal' form representation of a LOTOS specification consists of the process definitions, represented as Prolog objects in a 'flattened' name space, and the renaming data base that relates the internal names to their corresponding external names. In addition, it contains sort equations and rewriting rules data bases used by the ADT interpreter [FEH].

Every identifier in the LOTOS specification is renamed by a unique internal form to prevent ambiguities in the interpretation, since a specification may contain different identifiers with the same external name. The identifiers are: 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. The internal name of the main specification begins with 'sp'.

The following is a sample LOTOS specification, and its 'internal' form representation.

```
specification example[a1,a2]:noexit
type BITs is
sorts   Bit
opns        0   :-> Bit
        1   :-> Bit
endtype
behavior
a1?X:Bit;
a2?Y:Bit;
offer[a1,a2](X,Y)
where process offer [b1,b2](V:Bit, W:Bit):noexit :=
b1!V;
b2!W;
stop
[]
b2!V;
b1!W;
stop
endproc
endspec
```

The corresponding internal form representation for the **control** part is as follows:

```
proc(sp9, [g10,g11], [ ],
seq([g10([10]), [ [$,V4,v4,s12] ],[] ],
```

```
seq([g11([11]), [ [$,V6,v6,s12] ],[] ],
instance(p15,[g10,g11], [val(V4,v4),val(V6,v6)]))
).
proc(p15, [g7,g8], [ [V1,v1,s12],[V2,v2,s12] ],
choice(
[
seq([g7([15]), [ [#,val(V1,v1),s12] ],[] ],
seq([g8([16]), [ [#,val(V2,v2),s12] ],[] ],
stop([17]))),
seq([g8([19]), [ [#,val(V1,v1),s12] ],[] ],
seq([g7([20]), [ [#,val(V2,v2),s12] ],[] ],
stop([21]))
])
).
```

The renaming data base has the following form:

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

The first argument is the internal name of the identifier, while the second argument is its external name. The latter is enclosed in single quotes so that PROLOG will not interpret it as a PROLOG variable instead of an atomic name. This can happen when the name starts with an upper case character. The third argument is the internal name of the structure where the identifier is defined.

The renaming database of the above example is:

```
renm(sp9, 'example', []).
renm(g10, 'a1', [sp9]).
renm(g11, 'b2', [sp9]).
renm(p15, 'offer', [sp9]).
renm(g7, 'b1', [p15]).
renm(g8, 'b2', [p15]).
renm(v4, 'X', [sp9]).
renm(v6, 'Y', [sp9]).
renm(v1, 'V', [p15]).
renm(v2, 'W', [p15]).
renm(t13, 'BITs', [sp9]).
renm(s12, 'Bit', [t13]).
renm(o17, '0', [t13]).
renm(o18, '1', [t13]).
```

The external name of any object can be found directly by unification. For example we can obtain the external name for p15 by the following query:

```
renm(p15,X,_).
```

then X will be instantiated to 'offer', and the underscore '_' means that the third field is not required.

The process definitions are represented as PROLOG facts of the form:

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

where <proc-id> is the process name, <formal-gates> is the list of the process' formal gates,

<formal-pars> is a list of zero or more formal parameters, and <beh-exp> is the behaviour expression of the process.
Each formal parameter is represented as:

[<prolog-var>,<var-name>,<var-sort>]
where <prolog-var> is a unique PROLOG variable that will hold the value of the LOTOS variable where its internal name and sort are given in <var-name> and <var-sort> respectively.
<prolog_var> occurs in <beh_exp> so that when a value is assigned to it, all occurrences of the same <prolog-var> will have the same value (i.e. automatic substitution).
In the above example, the formal parameters of process **offer** have the following internal form:
[[V1,v1,s12],[V2,v2,s12]]
The following Prolog query:
proc(p15,GL,[[0,_,_], [1,_,_]], B).
will be unified with the internal form representation of the process **offer** and:
- **GL** will be instantiated to the list **[g7,g8]**
- **[[0,_,_], [1,_,_]]** will be unified with
[[V1,v1,s12],[V2,v2,s12]] that will cause the variables **V1**
and **V2** to be instantiated to **0** and **1** respectively.
- **B** will be instantiated to the behaviour expression of the
process **offer** with all the occurrences of **V1** and **V2** replaced
by **0** and **1** respectively, caused by the previous instantiation.
A variable can be instantiated by formal and actual parameter matching (unification), input, or by an assignment statement.
Internal actions are represented as:
i([<line_number>])
where <line_number> is the line number where the internal action appears in the source specification. <line_number> is also added to external actions and to the behaviour expression **stop**.
External actions have the following form:
[<gate>([<line_number>]), [<experiments>], [<predicate>]]
where:
<gate> is the internal form of the action's gate.
<experiments> is zero or more experiments.
<predicate> is the selection predicate of the action. It may
be empty.
Each experiment can have one of the following forms:
i) **[$,<prolog-var>,<var-name>,<var-sort>]**
representing the 'accept' experiments
ii) **[#,<expression>,<exp-sort>]**
representing the 'offer' experiments, where <expression> is the internal representation of a LOTOS expression in prefix format,and <exp-sort> is its sort.
Any variable in <expression> is represented by :
val(<prolog-var> , <var-name>)
so the name and value of the variable are kept together.
In the above example the actions:
a1?X:Bit

and
b2!V
are represented as:
[g10([10]), [[$,V4,v4,s12]],[]],
and
[g7([15]), [[#,val(V1,v1),s12]],[]],
respectively. They have no selection predicates.
The parameters of the behaviour **exit** are represented as events on the gate **exit** which represent the internal name of the gate δ.
The complete BNF of the 'internal' form representation of LOTOS specifications is given in Appendix E.

4.3 Behaviour Tree Semantics

The semantics definition of LOTOS, described in Chapter 2, is based on the concept of Transition Tree, where the branches represent all possible transitions and the nodes represent the states of the behaviour.

Unfortunately, by this semantics, in many cases an infinite number of transitions will result from a given behaviour. For example

B = **g1?X:Nat[X>5]; g2!X; stop**

allows the following infinite number of transitions:

B -g1 6-> g2!6; stop
B -g1 7-> g2!7; stop
B -g1 8-> g2!8; stop
..
..
..

Example 4.1

In the case of complex selection predicates, the transitions may be impossible to derive, e.g. **g?X:Nat ?Y:Nat [X4 + Y4 = 0]**.
For these reasons, we based our model on the concept of a Behaviour Tree, described earlier in Section 3.1, where the branches represent the possible "next actions" that the current behaviour can perform. The notation used for 'branch(B1, **a**, B2)' denotes that B1 can execute action **a** and the resultant behaviour will be $[t_1/x_1,..,t_n/x_n]$ B2, where t1,..tn are value expressions provided by the user for the variables of the action parameters $x_1,..,x_n$.
An action may consist of event parameters and predicates that restrict the values of such parameters. Because of the presence of variables and predicates, what is an infinite number of transitions in the Transition Tree may be a single branch in the Behaviour Tree. A branch in the Behaviour Tree can be explored by providing values for the action parameters that satisfy the associated predicates, if any, so that a transition and the resulting behaviour can be computed. Example 4.1 can then be represented by only one branch in the Behaviour Tree having the action **g1?X:Nat [X>5]**, which gives **g2!X; stop** as the behaviour of the subsequent node. This is denoted by
branch(B, g1?X:Nat [X>5], g2!X; stop)
To execute this action the user has to provide a value, say **N**, of type **Nat** such that **eval(N)** > **5**. Then the transition fired will be
B -g1 eval(N)-> g2!N;stop

which is one of the possible transitions in example 4.1.
Therefore, by the semantics of 'branch(B1,a,B2)', we then can obtain the semantics of a transition 'B1 -tr-> B2' using the following inference rule:

branch(B1,a,B2), execute(a, tr, [pv_1,..pv_n], [ap_1,..,ap_n])

B1 -tr-> [pv_1/ap_1,..,pv_n/ap_n]B2

where

ap_1,..ap_n are the variables of the action parameters,

pv_1,..,pv_n are the value expressions entered by the user, and

execute is a function that asks the user for pv_1,..,pv_n

4.4 Summation on Values Implementation

Another complicated situation is presented by the 'summation on values' construct where nondeterministic behaviours are generated by the 'choice' value parameter. The inference rule of such a construct can be defined by the Behaviour Tree semantics as:

if B = **choice** x:s [] B'

then

branch([t/x]B', **a**, B")

---

branch(B, **a**, B")

iff eval(t) $\in$ domain(s).

Note that if domain(s) is infinite, this rule corresponds to infinite number of inference rules. Since [t_1/x]B is equivalent to [t_2/x]B if t_1 = eval(t_2) (i.e. they have the same Transition Tree), then we can regroup all such behaviours into one. The condition on the above inference rule then becomes:

iff t $\in$ domain(s).

Therefore this inference rule is equivalent to:

branch([t_1/x]B' [] [t_2/x]B' [] ... [] [t_n/x]B', **a**, B")

---

branch(B, **a**, B")

where domain(s) = { t_1,t_2,...,t_n }

For example if B =

choice x:Bool **[]** g!x; p[g](x)

where domain(Bool) = { true, false },

then

branch(g!true;p[g](true) [] g!false;p[g](false), **a**, B")

---

branch(B, **a**, B")

The branches obtained are:

branch(B, g!true, p[g](true))

branch(B, g!false, p[g](false))

Now consider example 4.2 where the domain of the 'choice' variable is infinite:

B =

choice x:Nat []

[x mod 2 = 0] -> g!x?y; p[g](x,y)

Example 4.2

domain(Nat) = { 0, 1, 2, 3, ... },
then
branch(B2', **a**, B")
---
branch(B, **a**, B")
where
B' = [x mod 2 = 0] -> g!x?y; p[g](x,y)
and
B2' =
[0/x]B'
[]
[1/x]B'
[]
[2/x]B'
[]
[3/x]B'
[]
...
...

Since [t/x]B' is equivalent to **stop** when t in an odd natural number, then the following branches only can be generated:

branch(B, g!0?y, p[g](0,y))
branch(B, g!2?y, p[g](2,y))
branch(B, g!4?y, p[g](4,y))
...
...

The generation of this infinite number of branches is due to the summation of an infinite number of behaviours.

Unfortunately, generation of an infinite number of branches would be impossible in a simulator. For this reason, we limit the summation of behaviours to only one behaviour at a time, corresponding to a 'choice' value provided by the user.

The inference rule for
B = **choice** x:s [] B'
can then be defined as follows:
branch([t/x]B', **a**, B")
---
branch(B, **a**, B")
if t = eval(t') $\in$ domain(s)
where t' is provided by the user.

Therefore, every time a 'choice' is encountered during simulation, the user is asked to provide a 'choice' value.

The following problems may arise using the above implementation:

a) A non-exit recursion: applying the latter inference rule on
the behaviour of the process in example 4.3, the user will be

asked to provide ’choice’ values infinitely, due to the recursive call of the same behaviour.

```
p[g](x:Nat):noexit:=
    ( choice y:Nat [] p[g](y) )
    []
    g!x; stop
endproc
```

Example 4.3

b) In example 4.4, the user will be asked to provide a ’choice’ value for x, even if (s)he is only interested in simulating behaviour **g?y:Nat; B2**.

```
( choice x:Nat [] B’)
[]
g?y:Nat; B2
```

Example 4.4

c) The user may provide a ’choice’ value that causes guards to be evaluated to false, leading to an unspecified deadlock. (e.g. in example 4.2, any odd natural number leads to a deadlock).

d) The user may provide a ’choice’ value that causes an unspecified deadlock due to mismatching synchronized actions. In example 4.5, any ’choice’ value other than 2 causes a deadlock.

```
choice x:Nat [] g!x; B’
|[g]|
g!2; B"
```

Example 4.5

ISLA uses the concept of "pseudo-action" which requires a ’choice’ value. The user should keep in mind that this action is not a regular LOTOS action, and must not be considered in the generation of a test interaction sequence. This "pseudo-action" is marked by an asterisk in the menu of "next actions" and is performed on a dummy gate, called **’dummy’**. The inference rule for ’choice’ is replaced by the axiom:

branch(**choice** x:s **[]** B’, dummy?x:s, B’)

With this concept, problems (a) and (b) are solved. In example 4.3, the recursion is controlled because the following branches are produced:

branch(B, dummy?y:Nat, p[g](y))

branch(B, g!x, **stop**)

where B =

(**choice** y:Nat [] p[g](y))

[]

g!x; **stop**

In example 4.4, the user can now simulate behaviour **g?y:Nat; B2** without having to enter a ’choice’ value. The branches produced are:

branch(B, dummy?x:Nat, B’)

branch(B, g?y:Nat, B2)

where B=

(**choice** x:Nat [] B’)

[]
g?y:Nat; B2

To deal with problem (c), the user must first of all set a Check Point at the choice point. (S)he then can ask that all guards evaluated after a choice value is entered be listed, together with the results of their evaluation. In this way, it is possible to see if the choice value caused any guard to fail, and if so, the interpretation can be backtracked to the choice point, where another value can be selected.

Concerning problem (d), is the user's responsibility to look ahead in the resulting behaviour for synchronizations involving the 'choice' value.

To help the user in dealing with problems (c) and (d), we are currently working on an extension to the above implementation. This extension involves identifying these problems internally, and notifying the user if, due to a 'choice' value, guards were evaluated to false or mismatching synchronizations occurred. Also, it will give the user the opportunity of automatically backtracking the execution to enter a new value.

Another solution, involving a transformation of the specification to eleminate the choice constructs, is suggested in [EIJ].

In the following sections we define the Inference System that defines the semantics of LOTOS Behaviour Trees. We then transform this system into an executable Inference System. Finally we show how the latter system can be optimized by keeping the correctness of its definitions.

4.5 The Inference System

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

The notation branch(B1, **a**, B2) denotes that B1 can execute action **a** and the resultant behaviour will be $[t_1/x_1,..,t_n/x_n]$ B2, where t1,..tn are value expressions provided by the user for the variables of the action parameters $x_1,..,x_n$.

Defining the semantics of 'branch(B1,a,B2)', we then can obtain the semantics of a transition 'B1 -tr-> B2' using the following inference rule:

branch(B1,a,B2), execute(a, tr, $[pv_1,..pv_n]$, $[ap_1,..,ap_n]$)

B1 -tr-> $[pv_1/ap_1,..,pv_n/ap_n]$B2

where

$ap_1,..ap_n$ are the variables of the action parameters,

$pv_1,..,pv_n$ are the value expressions entered by the user, and

execute is a function that asks the user for $pv_1,..,pv_n$

4.5.1 Inference Axioms

The following rules generate the axioms:

a- Action Prefix

branch(a;B, a, B) is an axiom

EXAMPLE

branch(g1?X:Nat;g2!Succ(X);**stop**, g1?X:Nat, g2!Succ(T);**stop**)
is an axiom

branch(g?X:Nat[X>Succ(0)];**stop**, g?X:Nat[X>Succ(0)], **stop**)
is an axiom

b- Successful Termination

a) Without Value Passing

branch(**exit**, δ , **stop**)

is an axiom.

b) With value Passing

branch(**exit**(E_1,..,E_n), δ E_1,..,E_n, **stop**)

is an axiom

c- Summation on Values

branch(**choice** x:s [] B', **dummy**?x:s, B')

is an axiom.

4.5.2 Inference rules

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

a- Local Definition

branch([t_1/x_1,..,t_n/x_n] B', a, B")

---

branch(**let** x_1:s_1=t_1,..,x_n:s_n=t_n **in** B', a, B")

b- Guard

eval(P) = true, branch(B', a, B")

---

branch([P] -> B', a, B")

c- Choice

branch(B_1, a, B_1')

---

branch(B_1 [] B_2, a, B_1')

branch(B_2, a, B2,)

---

branch(B_1 [] B_2, a, B_2')

Or more general inference rule

branch(B_i, a, B_i')

---

branch(B_1 [] B_2 [] ... [] B_n , a, B_i')

$B_i \in \{ B_1, B_2, ..., B_n \}$

The implementation adopted the latter rule.

d- Hiding

branch(B', a, B"), name(a) $\notin \{g_1,..,g_n\}$

---

branch(**hide** g_1,..,g_n **in** B', a, **hide** g_1,..,g_n **in** B")

branch(B', a, B"), name(a) $\in \{g_1,..,g_n\}$

---

branch(**hide** g_1,..,g_n **in** B', **i**(a), **hide** g_1,..,g_n **in** B")

e- Nested

branch(B, a, B’)

---

branch((B), a, B’)

f- Parallel

Table 4.1 shows the resulting actions of all possible types of interactions between two processes. Similar rules apply when more than two processes are involved in an interaction.

process	process	synch.	interaction	resulting
A	B	condition	sort	interaction
$g!E_1$	$g!E_2$	$eval(E_1)$	value	$g!eval(E_1)$
		=	matching	
		$eval(E_2)$		
g!E	g?x:t	eval(E)	value	g!eval(E)
		$\in$	passing	and
		domain(t)		x = eval(E)
$g?x:t_1$	$g?y:t_2$	$t_1 = t_2$	value	g?[x,y]:t1
			generation	(i.e. a value
				is expected
				for x and y)

Table 4.1 Resulting Interactions

i) selected synchronization

branch(B_1, a1, B_1’), branch(B_2, a2, B_2’),

name(a1) = name(a2) $\in \{g_1,..,g_n, \delta\}$

---

branch(B_1 |[g_1,...,g_n]| B_2, a3, B_1’ |[g_1,...,g_n]| B_2’)

a3 is the resulting interaction (refer to Table 4.1).

branch(B_1, a, B_1’), name(a) $\notin \{g_1,..,g_n, \delta\}$

---

branch(B_1 |[g_1,...,g_n]| B_2, a, B_1’ |[g_1,...,g_n]| B_2)

branch(B_2, a, B_2’), name(a) $\notin \{g_1,..,g_n, \delta\}$

---

branch(B_1 |[g_1,...,g_n]| B_2, a, B_1 |[g_1,...,g_n]| B_2’)

ii) interleaving

branch(B_1 |[]| B_2, a, B’)

---

branch(B_1 ||| B_2, a, B’)

iii) full synchronization

branch(B_1|[g_1,..,g_n]|B_2, a, B')

---

branch(B_1 || B_2, a, B')

where {g_1,..,g_n} = G, is the union of all possible gates of B1 and B2.

g- Enable

branch(B_1, a, B_1'), name(a) $\neq \delta$.

---

branch(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)

branch(B_1, δ E_1,..,E_n, B_1')

---

branch(B_1 >> **accept** x_1:s_1,..,x_n:s_n **in** B_2, **i**, [E_1/x_1,..,E_n/x_n] B2

h- Disable

branch(B_1, a, B_1'), name(a) $\neq \delta$

---

branch(B_1 [> B_2, a, B_1' [> B2)

branch(B_1, δ E_1,..,E_n, B_1')

---

branch(B_1 [> B_2, δ E_1,..,E_n, B_1')

branch(B_2, a, B_2')

---

branch(B_1 [> B_2, a, B_1')

j- Relabelling

Note that the relabelling behaviour expression appears only in the dynamic semantics of the behaviour expressions as an additional expression, and it is not in the syntax of LOTOS.

branch(B', a, B''), name(a) $\notin$ {h_1,..,h_n}

---

branch((B')[g_1/h_1,..,g_n/h_n], a, (B'')[g_1/h_1,..,g_n/h_n]

branch(B', gE_1,..E_n, B''), g = hi (1 $\leq$ i $\leq$ n),

---

branch((B')[g_1/h_1,..,g_n/h_n], $g_i$$E_1$,..,$E_n$, (B'')[$g_1$/$h_1$,..,$g_n$/$h_n$]

k- Summation on Gates

branch((B')[g_i/g], a, B'')

---

branch(**choice** g **in** [g_1,..,g_n] [] B', a, B'')

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

l- par-Expression

branch(B'[g_1/g] op ... op B'[g_n/g], a, B'')

---

branch(**par** g **in** [g_1,...,g_n] op B', a, B")

is an inference rule where

$g,g_1,..,g_n$ are gate-variable instances,

op is a parallel-operator.

m- Process Instantiation

branch(([t_1/x_1,...,t_m/x_m]B)[g_1/h_1,..,g_n/h_n], a, B')

---

branch(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

n- stop

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

4.6 Inference System Implementation

The inference rules system is implemented using Prolog clauses of the form:

branch(B1,A,B2) :- cond1,...,condn.

where B1 is the given behaviour expression, A is an action that B1 can offer, and B2 is the resulting behaviour. A and B2 are to be determined by the inference rules by satisfying the sequence of conditions cond1,...,condn. Clauses of that form are called *rules*.

For instance the inference rule of the nested behaviour expression is defined by the following Prolog rule:

branch(nested(B),A,B2) :- branch(B,A,B2).

It can be read as "behaviour expression nested(B) can offer action A and transform to behaviour B2 if behaviour B can offer action A and transform to behaviour B2".

Axioms are implemented using clauses with no conditions on the right hand side, called *facts*. For example the axiom for action prefix can be defined by the following Prolog fact:

branch(seq(A,B),A,B).

It can be read as "behaviour seq(A,B) can offer action A and transform to B".

There is one or more inference rule for each behaviour expression.

Suppose we have the behaviour expression:

(i;stop)

whose internal representation is:

nested(seq(i,stop))

then the goal is

1- branch(nested(seq(i,stop)), A, B2).

Prolog tries to match (or unify) this goal with a head of a

defined clause by searching its database from top to bottom.

2- Clause found: branch(nested(B),A,B2)

Unification: B is instantiated to seq(i,stop)

3- New goal: branch(seq(i,stop),A,B2)

Clause found : branch(seq(A,B),A,B) by searching the database from top to bottom.

Unification: A is instantiated to 'i'

B is instantiated to ’stop’
and B2 is unified with B, it will also have the
value ’stop’
Then the goal branch(nested(seq(i,stop)), A, B2) succeeds with
A = ’i’ and B2 = ’stop’.

In the following section, Prolog clauses corresponding to the inference rules are defined one by one. That is, for every inference rule there is a corresponding Prolog clause. Then in the next section, we show how certain Prolog clauses are combined to optimize execution.

To avoid confusion, we have used the same notations used in section 4.3.1 such as B’ and B" where they are not acceptable by Prolog. In real implementation different names are used.

4.6.1 One-to-One Implementation

4.6.1.1 Axioms

a- action-prefix

branch(seq(A,B),A,B).

b- successful termination

branch(exit(A),A,stop).

where A has the form of a normal action on gate ’exit’ (internal name for δ), and the events are the exit-parameters (refer to the internal form).

c- summation on values

branch(distchoice(V,B’),[dummy,[$|V],[]],B’).

The generalized choice offers a pseudo action at gate **dummy**. This action does not interact with the environment. It is used only to generate one behaviour choice by entering a value for **V**. Refer to section 4.3 for more explanation.

4.6.1.2 Inference rules

a- Local-definition

```
branch(let(EQs, B’), A, B") :-
    asglet(EQs, B’, B2’),
    branch(B2’,A, B").
```

where asglet(EQs,B’,B2’) assigns the values of ’let’ in B’ giving B2’ (i.e. B2’ = $[t_1/x_1,..,t_n/x_n]$B’).

b- Guard

```
branch(guard([P], B’), A, B") :-
    eval(P,true),
    branch(B’, A, B").
```

c- Choice

```
branch(choice(BL),A,Bi’) :-
    member(Bi,BL),
    branch(Bi,A,Bi’).
```

where BL is a list of all behaviours involve in the choice (i.e. BL = {B1, B2, ..., B_n}).

member(Bi,BL) succeeds if Bi $\in$ BL, and fails otherwise. If Bi is a variable, which is the case in the above rule, member(Bi,BL) will instantiate B with the first element in BL, and if the execution backtracks to resatisfy member(Bi,BL), then Bi will be instantiated to the second element in BL and so on.

The above rule states that for every behaviour Bi $\in$ BL, if Bi can offer action A and transform to Bi’, then so can choice(BL).

d- Hiding

i- branch(hide(GL,B'), A, hide(GL,B")) :-
branch(B',A,B"),
\+common(GL,A). /* name(A) $\notin$ GL = {g_1,..,g_n} */

ii- branch(hide(GL,B'),i, B") :-
branch(B',A,B"),
common(GL,A). /* name(A) $\in$ GL = {g_1,..,g_n} */

where common(GL,A) succeeds if name(A) $\in$ GL, and fails otherwise. The '\+' operator is a negation, that is '\+goal' succeeds if 'goal' fails, and fails otherwise.

e- Nested

branch(nested(B),A,B') :-
branch(B,A,B').

f- Parallel

Since the interleaving synchronization '|||' is already defined in the internal form as a selected synchronization with empty gate list '|[]|', and the maximum synchronization is identified with the keyword 'maxsync' where a rendez-vous must occur on every action, then only three Prolog clauses are needed to cover the inference rules of the parallelism.

Rendez-vous synchronization between behaviour B1 and behaviour B2 must occur if one of the following is true:

i- B1 and B2 are composed with maximum synchronization
ii- there exist branch(B1,A1,B1') and branch(B2,A2,B2'), and
name(A1) = name(A2) $\in$ GL = list of synchronization gates

If none of the above conditions is true, then the next actions offered by B1 and B2 interleave. Prolog clauses then can be written as follows:

```
branch(parcomp(SYNC,B1,GL,B2), A,parcomp(SYNC,B1',GL,B2)):- branch(B1,A,B1'),
    SYNC \== maxsync, /* not a max. sync. */
    \+common(GL,A). /* name(A) ∉ GL */
branch(parcomp(SYNC,B1,GL,B2), A,parcomp(SYNC,B1,GL,B2')):- branch(B2,A,B2'),
    SYNC \== maxsync, /* not a max. sync. */
    \+common(GL,A). /* name(A) ∉ GL */
branch(parcomp(SYNC,B1,GL,B2), [G,ME,PD3],
                parcomp(SYNC,B1',GL,B2')):-
    branch(B1,[G,E1,PD1],B11'),
    branch(B2,[G,E2,PD2],B22'),
    valid(SYNC,G,GL), /* SYNC == maxsync or G ∈ GL */
    matching(E1,E2,ME,B11',B22',B1',B2'),
    combine(PD1,PD2,PD2).
```

where:

valid(SYNC,G,GL) succeeds if rendez-vous synchronization should occur, that is if SYNC = maxsync or G $\in$ GL.

matching(E1,E2,ME,B11',B22',B1',B2') matches event E1 with event E2 resulting in the matching event ME following table 4.1, and if there is value passing then the resulting behaviours become B1'and B2'.

g- Enable

```
/* no successful termination */
branch(enable(B1,ACPT,B2), A, enable(B1',ACPT,B2) :-
```

```
        branch(B1,A,B1'),
        \+common([exit],A).
    /* successful termination */
    branch(enable(B1,ACPT,B2), i, B2') :-
        branch(B1,[exit,V1,[]],B1'),
        pass_values[V1,ACPT,B2,B2').
```

where pass_values passes the values in 'exit' to B2 corresponding to the variables in ACPT, and resulting B2' (i.e. B2' = $[E_1/x_1,..,E_n/x_n]$ B2).

h- Disable

```
    branch(disable(B1,B2),A,disable(B1',B2)) :-
        branch(B1,A,B1'),
        \+common([exit],A). /* name(A) ≠ δ */
    branch(disable(B1,B2),[exit|EV],B1') :-
        branch(B1,[exit|EV],B1').
    branch(disable(B1,B2),A,B2') :-
        branch(B2,A,B2').
```

j- Relabelling

```
    branch(relabel(B',[FG,AG]), A, relabel(B",[FG,AG]) ):-
        branch(B',A,B"),
        \+common(A,FG). /* name(A) ∉ FG */
    branch(relabel(B',[FG,AG]),A',relabel(B",[FG,AG]) ):-
        branch(B',A,B"),
        common(A,FG), /* name(A) ∈ FG */
        relabels(A,A',[FG,AG]).
```

where

FG = $\{h_1, .., h_n\}$ is a list of formal gates

AG = $\{g_1, .., g_n\}$ is a list of actual gates

relabels(A,A',[FG,AG]) returns action A' with:

name(A') = i<t<h element in AG if name(A) = i<t<h element in FG

and the events of A' = events of A.

k- Summation on gates

```
    branch(distchoice(G,GL,B'),A,B") :-
        member(Gi,GL),
        branch(relabel([[G], [Gi]],B'),A,B").
```

can be read as "for Gi ∈ GL, G is relabelled by Gi for every action that may be performed by B' on the gate G".

l- par-expression

```
    branch(distpar(G,GL,OP,OPGL,B'), A, B") :-
        simplify_B(G,GL,OP,OPGL,B',B1'),
        branch(B1',A,B").
```

where 'simplify_B' simplifies the internal form of

par g **in** [g1,..,gn] op B'

to the internal form of

B'[g1/g] op ... op B'[gn/g].

m- Process instantiation

branch(instance(P,AG,AP), A, B') :-
proc(P,FG,FP,B),
pass_param(AP,FG,B,B2),
branch(relabel(B2,[FG,AG]), A, B').

where AG = {g1,..,gn} is a list of actual gates
AP = {t1,..,tm} is a list of actual parameters
pass_param passes the actual parameters to B corresponding
to the formal parameters, and resulting B2.
(i.e. B2 = $[t_1/x_1,..,t_m/x_m]$ B)

n- stop

there are no Prolog rules for stop.

4.6.2 Combining Rules for Effeciency Improvement

Using the above inference system causes repeated calls of theinference rules in the case where more than one inference rule is defined for one LOTOS construct. For example, in the case of the parallel composition, the recursive calls required by the first two inference rules (i.e. branch(B1,A,BR1) and branch(B2,A,BR2)) are also required by the third. For this reason, an early version of our interpreter ran quite slowly on large specifications. This problem was solved by combining such rules in one. In this way, the number of inference rules invocations in the case of parallel composition is reduced to half, and if parallel compositions are nested to N levels then the number of invocations will be reduced by a factor of two to the power N. The same situation applies to hiding, disable, enable and relabelling constructs whose semantics were defined by more than one inference rule each with repeated calls.

The following is the optimized implementation of such rules.

a- hiding

branch(hide(GL,B),A2,B2) :-
branch(B,A1,B2),
hide_or_not(A1,GL,A2).

where hide_or_not(A1,GL,A2) returns
A2 = A1 if name(A1) $\notin$ GL
A2 = i if name(A1) $\in$ GL

b- parallel

branch(parcomp(SYNC,B1,GL,B2), A, BR) :-
allsolutions(B1,S1),
allsolutions(B2,S2),!,
check_parallel([B1,S1],[B2,S2],SYNC,GL,A,BR).

where:

* allsolutions(B,S): returns a list S with all possible actions
that B can offer with their corresponding resulting behaviours
in the following form:
[[A1,BR1],...,[An,BRn]]
* check_parallel: checks the possible synchronization between the
actions offered by B1 and B2, and returns the proper
interaction A with the resulting behaviour BR. When
backtracking is required, check_parallel returns the next
possible interaction and the resulting behaviour.

c- enable

branch(enable(B1,ACPT,B2),A2,B3) :-
branch(B1,A1,BR1),
enable_exit(ACPT,A1,A2,BR1,B2,B3).

where enable_exit, depending on A1, returns A2 and B3.

d- disable

The first two inference rules of hiding required the same calls and they were combined to give:

branch(disable(B1,B2),A,B3) :-
branch(B1,A,BR1),
disable_exit(A,BR1,B2,B3)).

where disable_exit returns B3 = BR1 if A is not an exit, and returns B3 = disable(BR1,B2) otherwise.

The third Prolog rule for disable remains the same.

branch(disable(B1,B2),A,B3) :-
branch(B2,A,B3).

e- relabelling

branch(relabel(B',[FG,AG]),A',relabel(B",[FG,AG])):-
branch(B',A,B"),
relabels(A,A',[FG,AG]).

where

FG = $\{h_1, .., h_n\}$ is a list of formal gates

AG = $\{g_1, .., g_n\}$ is a list of actual gates

relabels(A,A',[FG,AG]) returns action A' with:

A' = A if name(A) $\notin$ FG

name(A') = i<t<h element in AG if name(A) = i<t<h element in FG

and the events of A' = events of A.

4.6.3 Line Numbers and Process Instantiations

In order to provide the user with more information on the progress of the simulation, the inference rules were modified to return two additional parameters other than the action and resulting behaviour expression. These are:

i - Line numbers of the offered action. Associating each action during simulation with the line numbers of the cooperating action offers in the source text helps the user to relate the results of the simulation with the text of the specification. For that reason the translator includes line numbers in the internal representation of actions. When the offered action is the result of synchronization of several actions, a list of all the line numbers of the synchronized actions is associated to it. For example, in the menu of next actions, the list of line numbers [LN1,...,LNn] states that its associated action is the result of synchronization of $action_1$ to $action_n$ that appear at line LN1 to line LNn respectively.

ii - Process Instantiations: The inference rules also accumulate all the process instantiations that led from the current behaviour expression to the derived action. This way the user can identify all the process invocations and their arguments. Two parameters are added to obtain the process instantiations, the first parameter is the current list of process instantiations, and is initially set to empty, and the second is the resulting list.

The complete list of inference rules is given in Appendix F .

Therefore for the current behaviour expression, say B, we can obtain a next action and its

resulting behaviour, action's line numbers, and the process instantiation, by the call:

branch(B,A,BR,LN,[],PI).

4.7 Menu of "next actions" construction

In Prolog the built-in predicate **bagof** can be used to obtain a list (or a bag), say S, of all possible next actions,their resulting behaviours, their line numbers and the process instantiations for a behaviour expression B by the call:

bagof([A,BR,LN,PI], branch(B,A,BR,LN,[],PI), S).

Unfortunately, in some cases the number of next actions of a behaviour expression of a process may be infinite. This happens in the following example:

P[a,b] :noexit:=

a;stop

[]

P[b,a]

In these cases, the **bagof** can not be applied. For this reason, a new predicate is created, called **allsolutions(B,S,MAX),** that obtains the same list **S** above but up to a maximum number of solutions, specified by **MAX**, which represents the maximum width of the behaviour tree. **MAX** is equal to 20 by default but this value can be changed by the user.

allsolutions forces backtracking to the inference system call trying to obtain new solutions until no more solutions are found or the number of solutions reaches **MAX**.

During simulation, **allsolutions** is applied on the current behaviour expression, then the next actions are shown on the screen in their external form associated with the order number in which they appear in the list **S**, and with their line numbers. When action **N** is to be executed the following steps are done by the simulator:

1- The **N**th record is obtained from the list **S**.

2- If the action requires values from the environment, then the user is asked to provide them.

3- If the values provided by the user are valid (i.e. have the right syntax and the right sorts) then they will be substituted in the attached selection predicate and the resulting behaviour expression.

4- If a selection predicate is associated with the action then it is evaluated. If the result of the evaluation is false, then the user is notified and is asked to provide new values (step 2). Otherwise, the executed action is saved in the history database and the resulting behaviour expression becomes the current behaviour expression. The menu of next actions will then be listed, and so on.

4.8 External Displays

We recall that ISLA runs on the internal representation of a specification. This representation has a prefix form, corresponding to the priority levels of the behaviour operators. For example, the following behaviour

a;b;stop [] c;stop [> d;stop

has the the following internal form

```
disable(
    choice(
        seq(<internal_a>,
            seq(<internal_b>,stop)),
        seq(<internal_c>,stop)),
    seq(<internal_d>,stop)).
```

since
priority(';') > priority('[]') > priority('[>')
The external representation of any process definition or any behaviour expression can be obtained from their internal forms, and the user never needs to be aware of the latter form. The output is justified corresponding to the priority levels of the operators. For example, the output representation of the above example is:

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

The external representation of a behaviour B is generated by the call:

show(B,OFFSET)
The operator of behaviour B is displayed at column OFFSET. The nested behaviours are displayed with the recursive call
show(NESTED_B,OFFSET+4)
and so on. OFFSET is equal to 0 initially.

4.9 Representation of History

All executed actions are saved in memory during simulation, yielding a tree that can be called the execution tree or the history tree of the simulated process. The nodes of this tree are saved in a database called **history** and contain five fields:

1- The last executed action number
2- The last executed action3- The line numbers
4- Process instantiations
5- Reference to the previous action

Suppose that, during simulation, the variables A_N, A, A_LN, PI, and FATHER_REF have the above arguments respectively. Then the current history tree node will be saved by the built-in predicate **assert** by the call:

assert(history(A_N,A,A_LN,PI,FATHER_REF), CURRENT_REF).

where **CURRENT_REF** is a unique system reference returned by the above call. CURRENT_REF will be used as the reference to the current node and will be saved in the history of all subsequent nodes as the father node reference and so on.

The reference to the root of an execution tree is saved in the clause **root(ROOT_REF)**. ROOT_REF is obtained by saving the initial behaviour, say INITIAL_BEH, by the call:

assert(INITIAL_BEH, ROOT_REF).

A new execution tree is created with a new **ROOT_REF**, when a new process, or the same process but with different actual parameters, is simulated.

Setting the history database as such, we can obtain the following information for a given node reference, say **REF**:

1- If the node associated with **REF** is not the root (i.e. no

root(REF) exist), then its history can be obtained by the
call:
clause(history(A_N,A,A_LN,PI,FATHER_REF),true, REF)
where **clause** is a built-in Prolog clause.
2- A son node can be obtained by the call:
clause(history(A_N,A,A_LN,PI,REF), true, NEW_REF)
that can be read as "which node has **REF** as the father reference?".

Each node (except the root node) has exactly one father node, but it may have several son nodes. All the son nodes can be obtained by forcing backtracking on the previous query. To obtain the root reference to all existing execution trees, backtracking should be forced on the call **root(ROOT_REF)**.

With the above information an execution tree can be traversed easily by using recursion and backtracking.

The following Prolog clause traverses, in preorder, the execution tree with the root reference **ROOT_REF** (comments are surrounded by '/*' and '*/') :

```
traverse(ROOT_REF) :-
                    /* obtain a son node */
    clause(history(A_N,A,A_LN,PI,ROOT_REF), true, NEW_REF)
    write(A),   /* print the action */
    traverse(NEW_REF),/* traverse son subtree */
    fail.       /* forces the backtracking */
                    /* to obtain next son node */
```

The following Prolog clause prints all existing execution trees:

```
traverse_all :-
    root(REF), /* obtain a root reference */
    traverse(REF), /* traverse the given tree */
    fail.       /* forces the backtracking */
                    /* to obtain next tree */
```

4.10 Check Points and Automatic Execution

Check points are saved in a database called **check_point** which has two parameters, the behaviour expression and the reference of the node obtained from saving the history of that node (CURRENT_REF in previous section). For example we can determine if a Check Point exists at an execution tree node that has the reference **REF** by the query:

check_point(Beh,REF).

that fails if no Check Point is found with the reference **REF**, and succeeds otherwise.

Check Points can be identified by the order number in which they appear in the database. These numbers are displayed in the history tree and the user can resume execution at any of these Check Points by providing its order number.

As explained in chapter 3, a Check Point can be saved in an external file by saving the initial process instantiation and the path history. By user choice, the behaviour expression of the Check Point can also be saved. In the case where the behaviour expression is not saved, an **automatic execution** is done to obtainit, starting by the same initial process instantiation. The action number and values are taken from the path history.

When the user requires the execution to back up to a specific point on the execution path,

the following steps are executed by ISLA:

1- Obtain the reference of the desired point, say **REF, from the** history database.

2- Check if there is a Check Point that is set at that node by the query

check_point(Beh,REF).

3- If this query succeeds then the current behaviour expression becomes **Beh** and the simulation restarts from there.

4- If this query fails, then the closest Check Point on the predecessor nodes in the tree is obtained and an automatic execution is done starting at that point until the desired point is reached.

5- If no Check Point is found on the predecessor path then an automatic execution is done starting at the initial process instantiation.

4.11 Symbolic Behaviour Tree

To compute the Behaviour Tree of a behaviour expression symbolically (see section 3.2.11.1), the user is required to provide the maximum width, which is the maximum number of next actions for each node, and the maximum depth of the tree. The predicates that are not evaluated to 'false' are assumed to be 'true' and will be shown in the Symbolic Behaviour Tree. These are the predicates that are evaluated to true or cannot be evaluated because they depend on values from the environment.

The algorithm to compute the Symbolic Behaviour Tree starting at a behaviour expression called **Beh** with maximum width **W** and maximum depth **D**, is:

1- **D2** = 0

2- while **D2** $\neq$ **D** do

3- For **Beh**, obtain the list **S** of all possible next actions and their resulting behaviour expressions but not exceeding **W**.

4- For every next action **A** and its resulting behaviour **RBeh** in **S**:

5- Print **A**

6- **Beh** = **RBeh**

7- **D2** = **D2** + 1

end while

In symbolic execution, a variable's value is represented by an expression that shows how the value is computed as a function of the input values. For example, an action that offers at gate **g** the sum of two values previously entered for two variables having the same name **x**, could be represented as:

g!(x@1+x@2)

Every variable in the Symbolic Behaviour Tree must be associated with a unique number to differentiate between variables with the same external name but correspond to different values. To understand the reasons for this, consider for example one path of the Symbolic Tree of the process **pop_machine** in Appendix C. If variables are not associated with unique numbers then the path would be shown as:

1 coin-in ?coin:COIN [51]
| 1 coin-in ?coin:COIN [51]
| | 2 buttons ?[button-name,button-name]:BUTTON
[ge(0+coin+coin,price(button-name))] [53,57]

Consider the selection predicate at line 4

[ge(0+coin+coin,price(button-name))]

in the expression **coin+coin** no distinction can be made between the first and the second **coin**, giving the impression that this expression is equivalent to **2*coin**. When numbers are added, the above path instead is shown as:

1 coin-in ?coin@1:COIN [51]
| 1 coin-in ?coin@2:COIN [51]
| | 2 buttons ?[button-name@1,button-name@2]:BUTTON
[ge(0+coin@1+coin@2,price(button-name@1))] [53,57]

in this case the expression **coin@1+coin@2** clearly states the sum of the variable **coin@1**, taken from the first action, and the variable **coin@2**, taken from the subsequent action.

A more extensive example of symbolic execution is shown in Appendix C (Part 13).

Chapter 5 Executing Large Specifications

In this chapter, we report on our experiences in simulating a full-sized LOTOS specification, the specification of the OSI Transport Service Provider [OSI2]. We refer readers to [LGH] for a more detailed report on these experiences.

The Transport Service specification of [OSI2] is modularized according to the constraint-oriented specification style. In this style, the specification consists of independent processes, where each process has its own constraints on the primitives. Constraints can relate to the order in which primitives can be established (expressed by LOTOS behaviour expressions) and to the possible values of parameters (expressed by selection predicates). The parallel composition of these processes is then specified [VIS]. The constraint-oriented style has become the most currently used style for LOTOS specifications.

5.1 Performance Characteristics and Useful Features

In our first experience in executing the TS specification, we used the one-to-one inference rules implementation of the interpreter described in Chapter 4. Unfortunately this implementation led to a disappointing result. For example, it took up to 6 minutes to obtain the initial actions offered by the specification. After the optimizations described in section 4.6.2, it took a surprising time of only 6 seconds to obtain the same actions. This was due to the fact that the parallel composition constructs are nested to several levels in the TS specification.

As mentioned in Chapter 1, ISLA is programmed in **Quintus Prolog** under **UNIX**. In these experiences, ISLA was used in a compilation mode on a **SUN 3/75** system. The performance obtained appears to be quite adequate for use in the "one-stepper" mode. When the interpreter is used for reachability analysis or for generation of large amounts of data, the performance becomes inadequate. For these reasons, we are envisaging more radical optimizations, namely in the inference rules implementation and value expression evaluator. Also research is being done on detecting repeatedbehaviour expressions and applying the congruent equivalences [LG].

ISLA has features that are very useful in simulating large LOTOS specification such as the TS specification, namely:

1- User Defined Constants: the display is considerably reduced.
For example a typical TS primitive is presented as an 8
characters constant instead of the actual 400 characters.
2- User Defined Sets: complex selection predicates are
automatically evaluated on sets of possibilities.
3- Check Points and Histories: alternative branches are easily
explored.
4- Saving Check Points in an external file: simulation can be
completed in more than one session.
5- Symbolic Execution: a portion of the behaviour tree can be
computed.

5.2 Tracing Methodology

In order to understand the practical uses of the interpreter, one should first understand the methodology of protocol software development to which LOTOS relates. Nowadays, protocols are usually specified informally. A formal specification is then obtained, and if the specification is in LOTOS, our interpreter can be used in several ways: to obtain sample execution sequences, to obtain test sequences to test implementation, etc.

In the following we shall concentrate on using the interpreter for checking the

correspondence between the formal specification and the informal one. While this is being done, the interpereter can also be used to check for common errors such as deadlocks in the formal or informal specification. This can be done by:

1- obtaining test sequences from the informal standard and checking if they can be accepted by the formal specification.

2- checking if accepted sequences by the formal specification are specified in the informal standard.

3- obtaining test sequences that are not specified in the informal standard and checking if they will also not be accepted by the formal specification.

Figure 5.1 **Conformance Testing by ISLA**

Three methods have appeared appropriate towards simulating large specifications that can be used within ISLA. These methods are described in the next sections.

5.2.1 One-Stepper method

This method corresponds to the basic operation of our interpreter. The user exercises a path in the behaviour tree by entering a sequence of primitives by hand. In this way the acceptable primitives can be identified at a given point, etc. To exercise another path, Check Points can be used as well as the back function described in Chapter 3. Although this method is very flexible and useful for users who are learning the behaviour of the specification, it is limited by the speed of the user in choosing the actions and entering the data. This method can be speeded up considerably by using shorthand constants and automatic evaluation of complex predicates on sets of values as mentioned in section 3.2.7. This method is illustrated in Fig. 5.2 and Fig. 5.3.

Figure 5.2 One-Stepper Method.

Figure 5.3 One-Stepper Method using Constants and Sets.

5.2.2 Using Symbolic Trees

This method involves examination of the symbolic trees. Since large trees are impractical to compute and read, and since they contain many uninteresting paths, a tree is computed to a certain depth, and the one-stepper method is then used to reach a specific leaf node in the tree. A tree is then computed from that node, etc. The main problems using this approach is that many of the branches are unfeasible because they contain contradictory conditions. For example, in our work of simulating the Transport Service, by combining the selection predicates of several actions, we would get contradictory conditions such as "IsTCONreq(x) and IsTCONind(x)" that requires x to be aconnection request and a connection indication at the same time. Since x is used symbolically (i.e. no value is assigned to it) such a condition, unfortunately, can not be evaluated, thus by our interpreter it is assumed to be true and the symbolic subtree of the following behaviour is computed.

In the figure below the method of using symbolic trees is shown. The bold arrows correspond to the one-stepper method.

Figure 5.4 Method of Using Symbolic Trees.

5.2.3 Using Testing Processes

This methodology involves creating "testing processes" to be run in parallel with the specification. Such processes contain simple sequences of actions with value parameters that will synchronize with the external actions in the specification. This way, all the variables (except the variables in the choice constructs) are instantiated and the user is not required to provide any values. For that reason, it is useful to obtain the symbolic tree of specification and its tester. Since

all variables are instantiated, due to the synchronization with the tester process, the tree is considerably reduced and can be called an ’execution tree’. By inspecting the execution tree, one can check whether the sequences of data submitted by the tester are accepted. One can create a library of testing processes according to several testing scenarios.

J_____ Chapter 6 Conclusion and Future Work

Chapter 6 Conclusions and Future Work

6.1 Conclusions

In this thesis, we have described in detail the LOTOS interactive simulator ISLA. We have presented LOTOS in a tutorial manner. We have shown how LOTOS inference rules can be modified in order to be implementable in a systematic way using Prolog clauses. We have also pointed out the fact that these rules, if implemented exactly as written, do not lead to an efficient implementation. These rules were optimized such that the execution time is considerably improved. ISLA is based on these optimized rules.

With the objective of making the simulator a useful tool for various purposes, possibly for testing and test sequence generation, classes of features are added to it. One of these classes concerns the step-by-step exploration of the behaviour tree, giving users the flexibility of exploring various paths. This can be done by setting Check Points at some execution points, allowing users to jump directly to any of these points to exercise another nondetermenistic choice, producing another execution path. These exercised paths are called an 'execution tree history'. The internal representation of the execution tree history can be used for automatic executions allowing simulation to backtrack to any point even if no Check Point is set at the desired point. Another feature is symbolic execution which allows to produce the symbolic behaviour tree of a given process. Since behaviour trees are often infinite, the user is required to provide the upper bounds of such trees.

We have used ISLA to prototype a full-sized constraint-oriented LOTOS specification: the OSI Transport Service (TS) provider. Executing a constraint-oriented specification involves executing several processes at once, each process posing its own constraints on the next actions. These constraints are expressed in terms of very complex value expressions, which need to be abbreviated in order to be understood. We have presented an implementation that helps users to find value expressions for actions, satisfying their corresponding constraints. These values are taken from user predefined databases. Unfortunately, a great number of these constraints involve contradictory conditions that cannot be satisfied by any value expression.

For users wishing to take full advantage of this interpreter, we have presented several methodologies that we have used in simulating the Transport Service specification.

A result of this experiment is the observation that the most effective specification style is not necessarily the best for execution. Therefore, if it becomes common practice to use simulators, such as ISLA, to execute specifications while they are being written, this may lead to changes in the generally practiced specification style.

In the Appendices, we have constructed a LOTOS specification and we have presented the usage of ISLA in simulating this specification.

6.2 Future Work

In section 4.4, we presented the problems related to the simulation of the 'summation on values' constructs, where the user is required to provide 'choice' values. The values may lead to situations that appear to the user as deadlocks, although they may simply be the consequence of inappropriate choice of values. We are currently working on identifying these problems internally, and notifying the user if, due to a 'choice' value, actions were not derived.

ISLA would course be inadequate for reachability analysis on large protocols or, more generally, generation of large amounts of data. For these purposes, we are envisaging more radical optimizations, namely in the value expression evaluator and in certain frequently used inference rules. We are also working on a system to derive optimized symbolic execution trees [LG] that

includes:

- The detection of repeated behaviours. In this way, the repeated behaviours can be indicated by loop references. Detecting all repeated behaviours is not decidable, however some heuristics to identify some cases of repetition are conceivable .

- The detection of obvious contradictory conditions. These conditions can never be evaluated to 'true', therefore their resulting execution subtrees should be eliminated. Of course, detecting all contradictory conditions is also an unsolvable problem.

- The simplification by congruence rules. This simplification does not in any way change the semantics of the specification.This is done by identifying some obvious simplifications since, again, in general this is an undecidable problem.

The optimized symbolic execution trees can be used as a basis for the derivation of test suites.

We are in the process of implementing two graphical tools for LOTOS. The first tool is a graphic simulator using only the mouse. The other tool is a graphical representation of LOTOS behaviours. The two tools can be combined producing a complete graphical simulator for LOTOS. These tools are being implemented in X-Window with C language on a SUN workstation and they are based on ISLA by using an interface between C and Prolog.

Appendix A: LOTOS Behaviour Expressions' Syntax

J_____ **Appendix A: LOTOS Behaviour Expressions' Syntax**

LOTOS behaviour expressions	
g d_1 ... d_n[P]; B	action prefix
where	
d_i = !t_i or **?x_i:s_i**	
i; B	internal action prefix
exit	successful termination
or	
exit(E_1,..., E_n)	
where	
E_i is a term or	
E_i = **any s_i**	
let x_1=t_1,..,x_n=t_n in B	local definition
choice $g_=$ in [g_1,..,g_n] [] B	summation on gates
choice x:s [] B	summation on values
par g in [g_1,..,g_n] op B	par
hide g_1,..,g_n in B	hiding
B1 >> accept	enable
x_1:s_1,..,x_n:s_n in B2	
B1 [> B2	disable
	parallel
B1 ||| B2	(interleaving)
B1 || B2	(full synchronization)
B1 |[g_1,..,g_n]| B2	(selected synchronization)
B1 [] B2	choice
[Guard] -> B	guarded
stop	deadlock (inaction)

| **(B)** | nested |
+---------------------------------+-------------------------+
| **$p[g_1,..,g_n](t_1,..,t_n)$** | process instantiation |
+---------------------------------+-------------------------+

J**_____ Appendix B: Construction of a LOTOS specification**

Appendix B: Construction of a LOTOS specification

The following specification models a ’pop machine’ that accepts coins (dollars and quarters) and delivers the desired pop by pushing the corresponding button. The general idea of this example is taken from the ’vending machine’ by Bolognesi and Brinksma [BB].

The specification consists of the following processes: ’get-money’, ’deliver’, and ’pop-machine’.

The process ’get-money’ keeps accepting coins (dollars and quarters) from the user, at gate ’coin-in’ until he/she pushes a button of a specific pop, at gate ’buttons’. The button may only be pushed when the user has entered enough money to cover the price of the pop. Then the process will terminate successfully.

```
process get-money[coin-in,buttons](amount:COIN) :exit:=
        coin-in?coin:COIN;
        get-money[coin-in,buttons](amount + coin)
[]
        buttons?button-name:BUTTON [amount ge price(button-name)];
        exit
endproc
```

The process ’deliver’, after accepting a choice of pop at gate ’buttons’, will offer the pop in the drawer, specified by gate ’drawer’. Unfortunately, there exists a little devil that can disallow the machine to deliver the pop, and this is specified by the internal event ’i’. In either case the process will terminate successfully.

```
process deliver[buttons,drawer]:exit:=
buttons?button-name:BUTTON ;
( drawer!pop(button-name) ; exit
[]
i ; exit
         )
endproc
```

The overall action of ’pop-machine’ can then be described as follows:

```
process pop-machine[coin-in,buttons,drawer] :noexit :=
(
get-money[coin-in,buttons](0 of COIN)
|[buttons]|
deliver[buttons,drawer]
)
        >> pop-machine[coin-in,buttons,drawer]
endproc
```

The processes ’get-money’ and ’deliver’ interact among themselves and with the user at gate ’buttons’, and the interaction may occur only after the user has entered at least the price of the pop. When the interaction does occur, the pop may be offered by the process ’deliver’ or may be taken by the devil. After this, the pop machine will be ready for the next consumer.

The complete specification is listed below with the data types that describe the sorts COIN,BUTTON, and POP. The sort COIN is defined in terms of the natural number sort 'Nat' that is defined in the standard library.

```
library Boolean , NaturalNumber endlib
type coin-type0 is NaturalNumber renamedby
sortnames COIN for Nat
endtype
type coin-type is coin-type0
opns quarter , dollar : -> COIN
eqns ofsort COIN
        dollar = Succ(Succ(Succ(Succ(0)))) ;
        quarter = Succ(0) ;
endtype
type button-type is
sorts BUTTON
opns Milk-button , Pepsi-button ,
            Coke-button , V8-button : -> BUTTON
endtype
type pop-type is button-type , coin-type
sorts POP
opns Milk , Pepsi , Coke , V8 : -> POP
pop : BUTTON -> POP
price : BUTTON -> COIN
eqns forall x: BUTTON
        ofsort POP
            pop(Milk-button) = Milk; pop(Pepsi-button) = Pepsi ; pop(Coke-button) = Coke;
pop(V8-button) = V8 ;
        ofsort COIN
         price(Milk-button) = quarter + quarter ;
price(Pepsi-button) = quarter + quarter + quarter;
price(Coke-button) = quarter + quarter + quarter;
         price(V8-button) = dollar ;
endtype
behaviour
        pop-machine[coin-in,buttons,drawer]
where process pop-machine[coin-in,buttons,drawer] :noexit :=
(
get-money[coin-in,buttons](0 of COIN)
|[buttons]|
deliver[buttons,drawer]
)
>> pop-machine[coin-in,buttons,drawer]
endproc
process get-money[coin-in,buttons](amount:COIN) :exit:=
coin-in?coin:COIN ;
        get-money[coin-in,buttons](amount + coin)
```

```
[]
buttons?button-name:BUTTON [amount ge price(button-name)];
          exit
endproc
process deliver[buttons,drawer]:exit:=
buttons?button-name:BUTTON ;
( drawer!pop(button-name) ; exit
[]
i ; exit
          )
endproc
endspec
```

J_____ **Appendix C: Simulating a LOTOS specification using ISLA**

Appendix C: Simulating a LOTOS specification using ISLA

In this section, a simulation of the pop machine specification constructed in the previous section is given.

- Comments are preceded by the symbol '%'.
- What is displayed by ISLA is shown in bold face.
- The user input is underlined

```
% This is a comment
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 1 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
LOTOS: isla
| ?- isla.
% The first command executes prolog and charges the isla program,
% and the second command executes isla from prolog.
% After these two commands the main menu will be displayed as
% shown.

============================================================
ISLA
Interactive System for Lotos Applications
<1> (l) List LOTOS specifications '.l'
<2> (c) Compile a LOTOS specification '.l'
<3> (p) List compiled LOTOS specifications '.pl'
<4> (x) Execute a compiled LOTOS specification '.pl'
<5> (cp) List available Check Points '.cp1' & '.cp2'
<6> (xcp) Execute a Check Point '.cp1' & '.cp2'
(pwd) Show current directory
(cd) Change current directory
(ls) List directory
(vi) Text editor
<m> This Menu <h> Help Menu
<exit> Quit ISLA <u> Unix system call
============================================================
% To execute a LOTOS specification, the user should compile it
% then execute it.
% compilation
M:Choose one command or 'm' or 'h' for help ==> c
Enter file name or hit <RETURN> to continue ==> pop-machine

LOTOS compiler 87/2.1
(C) University of Ottawa - Protocols Research Group - 1987
syntax and semantic analysis of specification pop-machine.l
... done
```

make output pop-machine.pl
load library /usr/uotcsih/dept/lotos/lib/dis.x ... done
... done

% execution
M:Choose one command or 'm' or 'h' for help ==> *x*
Enter file name or hit <RETURN> to continue ==> *pop-machine*

[consulting /usr/uotcsih/dept/lotos/exmpls/pop-machine.pl...]
[pop-machine.pl consulted 3.967 sec 16,960 bytes]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 2 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% A listing of all the processes will be displayed.
% One of them can be executed by giving its corresponding number
% then the actual gates (optional) and the actual parameters
% (if any).
% The formal gates are taken as the actual gates when the user
% does not supply the latter (i.e. see below).

The available processes are:
[1] system[coin-in,buttons,drawer]()
[2] pop-machine[coin-in,buttons,drawer]()
[3] get-money[coin-in,buttons](amount:COIN)
[4] deliver[buttons,drawer]()
P:Choose one command or 'lp' or 'h' for help ==> *1*
Enter actual process gates for execution ==> system
% The interpreter then applies the inference rules on the
% behaviour of the given process and lists a menu of all
% possible actions.
% Each action is associated with: a unique number, the line
% numbers where the action is found in the specification (in
% square brackets), and eventually a list of selection
% predicates.
% Note also that action <2>, which is an interaction between two
% processes, has two variable names between square brackets after
% the question mark. These are the names of the two variables
% that will be assigned the same value in the two interacting
% processes as a result of the action. **bh1** and **bh2** are the
% behaviours resulting from action <1> and <2> respectively. The
% user can request to see them by typing **bh1** or **bh2**.
% Refer to section 3.2.2 for the exact action menu's format.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 3 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

============================== system ==============================
CP/MANUAL W/20 Level/0 Path/[]
==
<1>- coin-in ?coin:COIN ----> bh1 [51]
<2>- buttons ?[button-name,button-name]:BUTTON
[ge(0,price(button-name))] ----> bh2 [53,57]
==
% An action can be fired by giving its associated number. If the % action requires a choice of values, then the user should
% provide them.
% For example, action # 1 requires a value for 'coin' of sort
% COIN. The command 'av 1' can be issued to provide all
% corresponding valid expressions.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 4 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
A:Choose one command or 'la' or 'h' for help ==> *av 1*

COIN => COIN*COIN COIN => dollar
COIN => 0 COIN => **(COIN,COIN)
COIN => price(BUTTON)COIN => Succ(COIN)
COIN => +(COIN,COIN) COIN => quarter
BUTTON => Coke-button BUTTON => Milk-button
BUTTON => Pepsi-button BUTTON => V8-button
% e.g. The expression 'price(Coke-button)' is a valid value of
% sort COIN.
% The user can define his/her own constants that can be used
% later as shorthands.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 5 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

A:Choose one command or 'la' or 'h' for help ==> *cstd*
Enter your constants in the form <$CONST-NAME> = <EXPR>
At the end hit <RETURN>

|: *$25 = quarter*
-> Ok!!!
|: *$50 = +($25,$25)*
-> Ok!!!
|: *$75 = +($50,$25)*
-> Ok!!!
|: *$coke$ = price(Coke-button)*
-> Ok!!!
|:

% When choosing action 1, any value expression or sort COIN or % any of the above constants can be used as a value for 'coin'.
A:Choose one command or 'la' or 'h' for help ==> *1*
Enter a value for: coin:COIN => *$50*
% The actions offered by the resultant behaviour will then be
% displayed.
% Action 2 bellow, is an interaction between process 'get-money'
% and 'deliver' on the gate 'buttons'.

============================== system ==========================
CP/MANUAL W/20 Level/1 Path/[1]
Events [1] coin-in ?$50:COIN
==
<1>- coin-in ?coin:COIN ----> bh1 [51]
<2>- buttons?[button-name,button-name]:BUTTON
[ge(0+$50,price(button-name))] ----> bh2 [53,57]
==
A:Choose one command or 'la' or 'h' for help ==> *2*
The existing predicate(s) for this action:
[ge(0+$50,price(button-name))]
Enter a set or a value for:[button-name,button-name]:BUTTON =>
Coke-button
% not enough money to obtain a coke
Predicate evaluated to false
Do you like to see the trace of the evaluation ? (n/y) ==> *n*
Enter a set or a value for:[button-name,button-name]:BUTTON =>
Milk-button
Predicate evaluated to true
% At this point the user has pushed the button for 'Milk' and
% the machine has accepted it because the user has already
% entered enough coins to cover the price of the milk. The
% machine is going to deliver the milk.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 6 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A 'check point' can be set at any time during simulation
% by the command 'cpset'. We can go back to that point at any% time later.

============================== system ============================== CP/MANUAL W/20 Level/2 Path/[1,2]
Events [1] coin-in ?$50:COIN
[1,2] buttons ?Milk-button:BUTTON
==
<1>- drawer !Milk:POP ----> bh1 [59]
<2>- i (specified explicitly) ----> bh2 [61]
==

A:Choose one command or ’la’ or ’h’ for help ==> *cpset*
Ok!!!
% The pop may be delivered in the drawer or the devil may not
% allow it. Assume that the devil did not interrupt.

A:Choose one command or ’la’ or ’h’ for help ==> *1*
passed evaluated value: ==> Milk
========== system ========== CP/
MANUAL W/20 Level/3 Path/[1,2,1]
Events [1] coin-in ?$50:COIN
[1,2] buttons ?Milk-button:BUTTON
[1,2,1] drawer !Milk:POP
==========
<1>- i (enable: exit) ----> bh1 [54,59]
==========
% The above internal event passes the execution to the process
% ’pop-machine’ where the operation restarts.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 7 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The command ’cpl’ lists all available checkpoints where they
% are represented by their action numbers path. The user is able % to jump directly to any check point by giving its associated
% number.
A:Choose one command or ’la’ or ’h’ for help ==> *cpl*
The available Check Points are:
-<1>- [1,2]
A:Choose one command or ’la’ or ’h’ for help ==> *go1*
% Execution returns to the only available checkpoint.
========== system ========== CP/
MANUAL W/20 Level/2 Path/[1,2]
Events [1] coin-in ?$50:COIN
[1,2] buttons ?Milk-button:BUTTON
==========
<1>- drawer !Milk:POP ----> bh1 [59]
<2>- i (specified explicitly) ----> bh2 [61]
==========
A:Choose one command or ’la’ or ’h’ for help ==> *2*
% This time, the user has decided to follow a different execution
% path: the devil has blocked the delivery .

Internal event is executed
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 8 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

========== system ========== CP/
MANUAL W/20 Level/3 Path/[1,2,2]
Events [1] coin-in ?$50:COIN
[1,2] buttons ?Milk-button:BUTTON
==========
<1>- i (enable: exit) ----> bh1 [54,61]
==========
A:Choose one command or 'la' or 'h' for help ==> *level1*
% the command 'level' allows user to restart the simulation at
% any preceding level on the execution path.

========== system ========== CP/
MANUAL W/20 Level/1 Path/[1]
Events [1] coin-in ?$50:COIN
==========
<1>- coin-in ?coin:COIN ----> bh1 [51]
<2>- buttons ?[button-name,button-name]:BUTTON
[ge(0+$50,price(button-name))] ----> bh2 [53,57]
==========
A:Choose one command or 'la' or 'h' for help ==> *1*
Enter a value for: coin:COIN => *$25*
========== **system** ==========
CP/MANUAL W/20 Level/2 Path/[1,1]
Events [1] coin-in ?$50:COIN
[1,1] coin-in ?$25:COIN
==========
<1>- coin-in ?coin:COIN ----> bh1 [51]
<2>- buttons?[button-name,button-name]:BUTTON
[ge(0+$50+$25,price(button-name))] ----> bh2 [53,57]
==========
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 9 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% When an action requires value expressions to satisfy a
% selection predicate, the user can assign a set of values for
% any required value, then ISLA reports on the evaluation of the
% predicate using every possible combination of values taken
% from the sets. A set can be defined by the command 'setd' and
% should contain values of one specific sort.

A:Choose one command or 'la' or 'h' for help ==> *setd*
Enter the sort or <RTN> to quit ==> *BUTTON*
Enter the set name or <RTN> to quit ==> *button*

Enter your data expressions

At the end hit <RETURN>
|: *Milk-button*
Ok!!!
|: *Pepsi-button*
Ok!!!
|: *Coke-button* **Ok!!!**
|: *V8-button*
Ok!!!
|:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 10 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
A:Choose one command or 'la' or 'h' for help ==> *2*
The existing predicate(s) for this action : [ge(0+$50+$25,price(button-name))]
Enter a set or a value for:[button-name,button-name]:BUTTON => *set(button)*
1> Milk-button ==> true
2> Pepsi-button ==> true
3> Coke-button ==> true
4> V8-button ==> false
t<N> to trace, c<N> to choose or l to relist or <RTN> ? *c3*
% Therefore, with the amount of money that the user has
% entered, he/she can choose any pop except 'V8'.
% The user can choose any one of these values to be the desired
% value for simulation. In this case a Coke has been chosen.
========== **system** ========== **CP/**
MANUAL W/20 Level/3 Path/[1,1,2]
Events [1] coin-in ?$50:COIN
[1,1] coin-in ?$25:COIN
[1,1,2] buttons ?Coke-button:BUTTON
==
<1>- drawer !Coke:POP ----> bh1 [59]
<2>- i (specified explicitly) ----> bh2 [61]
==
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 11 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The command 'bh0' prints the current behaviour expression
% (i.e. the behaviour that offered the above actions).
A:Choose one command or 'la' or 'h' for help ==> bh0
Output to (s)creen or to a (f)ile or (q) ==> *s*
exit
|[buttons]|
(
drawer !pop(Coke-button):POP ; exit
[]
i ; exit

)
>> pop-machine[coin-in,buttons,drawer]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 12 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The command 'hall' displays all the actions taken so far in
% the current simulation session in the form of a tree.
% It also shows the number and position of all checkpoints on
% the left hand side, and the current simulation point is marked
% by the symbol '***'.

A:Choose one command or 'la' or 'h' for help ==> *hall*
Show process instantiations (n/y) ==> *n*
Output to (s)creen or to a (f)ile or (q) ==> *s*

```
cp |
---
| system[coin-in,buttons,drawer]
|
|..1 coin-in ?$50:COIN [51]
1 |....2 buttons ?Milk-button:BUTTON
[ge(0+$50,price(Milk-button))] [53,57]
|......1 drawer !Milk:POP [59]
|......2 i (specified explicitly) [61]
|....1 coin-in ?$25:COIN [51]
|******2 buttons ?Coke-button:BUTTON
[ge(0+$50+$25,$price)] [53,57]
A:Choose one command or 'la' or 'h' for help ==> q
Back to process menu (n/y) ? y
% leaving the simulation phase

The available processes are:
[1] system[coin-in,buttons,drawer]()
[2] pop-machine[coin-in,buttons,drawer]()
[3] get-money[coin-in,buttons](amount:COIN)
[4] deliver[buttons,drawer]()
P:Choose one command or 'lp' or 'h' for help ==> ptree1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 13 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The command 'ptree' displays the behaviour tree of a process,
% where the execution is done symbolically without the values
% from the environment. In other words, variable values are
% denoted by expressions that may involve variable names. Since
% different variables may have the same name, a unique number is
% attached to every distinct variable name.
% To limit the size of the tree, the maximum width and length of
% the tree should be given.
Maximum execution depth (default = 5) ==> 3
Show complete actions or only gates (a/g) ==> a
Show process instantiations (n/y) ==> n
Output to (s)creen or to a (f)ile or (q) ==> s

1 coin-in ?coin@1:COIN [51]
| 1 coin-in ?coin@2:COIN [51]
| | 1 coin-in ?coin@3:COIN [51]
| | 2 buttons ?[button-name@1,button-name@2]:BUTTON
                [ge(0+coin@1+coin@2,price(button-name@1))] [53,57]
| 2 buttons ?[button-name@1,button-name@2]:BUTTON
            [ge(0+coin@1,price(button-name@1))] [53,57]
```

| | 1 drawer !pop(button-name@1):POP [59]
| | 2 i (specified explicitly) [61]
2 buttons ?[button-name@1,button-name@2]:BUTTON
[ge(0,price(button-name@1))] [53,57]
| 1 drawer !pop(button-name@1):POP [59]
| | 1 i (enable: exit) [54,59]
| 2 i (specified explicitly) [61]
| | 1 i (enable: exit) [54,61]
A:Choose one command or 'la' or 'h' for help ==> *exit*
[End of Prolog execution]
LOTOS:

Appendix D: ISLA's Commands

Note : In any command :
- The part in square brackets "[]" is optional,
- The part in "<>" is a number and should be entered by the user.

Remark: All expressions are shown in prefix form, and they should also be entered in prefix (even if they are defined as infix).

---------- div_command ----------

h[elp] - this screen or some information about a topic
options - simple list of options in use
? - simple list of all commands
screen[<N>] - sets the displayed screen lines to <N>. Default
N = 20.
cw[<N>] - (change width),To change the Width of execution.
Default N = 20.
noclear - the screen will not be cleared during execution.
Useful for printing the whole execution session.
clear - the screen will be cleared during execution.
(Default)
q - goes back to process menu
qm - goes back to main menu
exit - leaves ISLA.

---------- info_on_bh ----------

<N> - Executes action number <N>.
la[f] - (List Actions) relist the current offered actions
"laf" to save this list on a file
bh[l]<N> - (Behaviour), prints behaviour expression number <N>.
To print the original (source) behaviour enter
'bh0' .
.bh prints only the first actions,
.bhl prints all behaviour except recursive calls.
tree[<N>] - generates the symbolic tree for process number <N>.
It does an automatic execution for the given
behaviour with no real evaluation (symbolically).
The default dimension of the tree can be changed.
Default : <N> = 0 ,(for the behaviour that fired the
current actions).
pred<N> - print guards evaluated to get the action number <N>
pi<N> - (process instantiations), shows all the process
invocations between action <N> and the previous
action.
how - trace the inference rules which gave the currents

actions.
stat - print the number of inferences rules used to get the actions.
av <N> - (parameters values), prints the valid values of the parameters of action number <N>.
?path - print the current path
internal - to skip some internal actions
.enable .distchoice
.internal action .internal_matching
(not implemented)
listinternal - to list these skipped actions when obtaining a new screen
[no]listnomatch - to list not matching actions when obtaining a new screen
[no]listevents - to list external action who lead to the actual point when obtaining a new screen (by default)
[not]ignore<ln> - to ignore certain actions defined at <ln> = LineNumber
.i specified explicitly
. choice values
listignored - to list lines ignored
[no]listignore - to list ignored actions when obtaining a new screen
---------- info_on_process ----------
<N> - To execute a process, enter the number associated with the process. Then the actual gates (optional), and the actual parameters (if any) are entered.
Ex: [G1,G2](succ(0),true)
Note:- the number of actual gates (if entered) should be equal to the number of the formal gates.
- the number of actual parameters (if any) should be equal to the number of the formal parameters, and they should be of the same types.
lp - (list all processes), lists all available processes with their correspondent numbers.
pv <N> - (parameter values), prints the valid values of the parameters of process number <N>.
pp[l]<N> - (pretty print), displays process number <N>.
sptree<N> - (process tree), generates the static definitions of the processes.
ptree<N> - generates the symbolic tree for process number <N>. It does an automatic execution for the given behaviour with no real evaluation (symbolically).
The default tree's dimension can be changed.
The output tree can be directed into a file.

---------- check_points and history ----------
auto - (automatic check point setting),check points will
be saved automatically on every execution point.
manual - (manual check point setting), check point can be
set only manually (see 'scp' & 'cp' commands).
save - saves the current point of execution for later
execution.
Here you will have the option of saving the whole
behaviour of the current point (MODE 1), or only
the history (MODE 2).
When resuming execution at that point using the
saved file, in MODE 2 an automatic execution will
be done to obtain the desired behaviour.
set - sets a check point for the current execution point.
go<N> - Restart execution at checkpoint number <N>.
back[<N>] - goes back <N> levels in the execution path.
Default N = 1.
If no CheckPoint is set at that level, a CheckPoint
will be computed starting from the closest
CheckPoint above the desired level, or from the
root if no CheckPoint exist in the execution path.
level <N> - goes to level <N> in the execution path.
If no CheckPoint is set at that level, a CheckPoint
will be computed starting from the closest
CheckPoint above the desired level, or from the
root if no CheckPoint exist in the execution path.
rmcp<N> - Removes checkpoint number <N>.
prcp<N> - Displays the behaviour of checkpoint number <N>.
save <N> - Saves checkpoint number <N> in a file, for later
execution.
Here you will have the option of saving the whole
behaviour of the Check Point (MODE 1), or only the
history (MODE 2).
When resuming execution at that point using the
saved file, in MODE 2 an automatic execution will
be done to obtain the desired behaviour.
'sv' can be used instead of 'save'.
hcp<N> - Prints all the actions that were taken up to
checkpoint number <N>.
tcp<N> - Prints the symbolic tree starting at checkpoint
number <N>.
ph - (point history), lists all the actions that lead
to the current execution point.
th - (tree history), lists all the actions taken so far
for the current initial call (initial process
call).

fh - (forest history),lists all the actions taken so far
for all the initial process calls of the same
specification.
---------- infos on evaluations ----------
trace[off|on] - to set automatic or manual (default) trace of
evaluations
le - to list the evaluations done to obtain the
currents actions
---------- use of values, defined constants and sets --------
vs - (valid sorts), prints all valid sorts used in the
current specification.
vo - (valid operators),prints all valid operators used in
the current specification.
cstd - Allows users to define their own constants.
cstl - lists all user defined constants.
cstrm[] - clears all (or one) user defined constants from
memory.
cstf - gives the user the following commands:
l)oad : to load a specific user defined constants
file
s)ave : to save all user defined constants in the
memory to a file
t)ype : type an existing user defined constants
file on the screen
d)ir : shows all available user defined constants
files
setd - Allows users to define their own sets of expressions or constants of same sorts
setl - lists all user defined sets in memory.
setc - lists content of one user defined set.
setrm - clears all (one) user defined set from memory.
setf - gives the user the following commands:
l)oad : to load a specific user defined set file
s)ave : to save one user defined set in the memory
to a file
t)ype : type an existing user defined set file on
the screen
d)ir : shows all available user defined sets files
cdtd - to set contradictory predicate
example: IsTReq(@1:TSP) # IsTInd(@1:TSP)
cdtl - lists all user defined contradictory predicate.
cdtrm - clears all (or one) user defined contradiction from
memory.
cdtf - gives the user the following commands:
l)oad : to load a specific user defined
contradictions file
s)ave : to save all user defined contradiction in

the memory to a file

t)ype : type an existing user defined contradictions file on the screen

d)ir : shows all available user defined contradictions files in the working directory

J_____ **Appendix E: Internal Form Representation**

Appendix E: Internal Form Representation

PROLOG REPRESENTATION OF LOTOS SPECIFICATIONS

1) Renaming function

Problems

- Identifiers with a same name that define different objects must be given different names because of the absence of scoping in Prolog.
- Prolog treats differently identifiers that begin with capital letters
- Variables (beginning with a capital letter) have a dynamic scope local to the inference rule in which they appear. There is no way to get back the name of a variable.

Solutions

- each object is given a distinct number N, and a one or two letter string S defined as follow:

sp for specification identifiers
p for processes
g for gates
s for sorts
o for operations
v for values

The new name is now obtained by appending N to S.

- The mapping between the real name and the internal one can be obtained from a symbol table of that form:

rename (new_name , old_name , path).

example:

'val' (v3) declared in process 'proc' (p2), declared in specification spec (sp1) will lead to the following:

rename (v3 , 'val' , [p2, sp1]).
rename (p2 , 'proc' , [sp1]).
rename (sp1 , 'spec' , []).

- Note that values must begin with a capital so that it can be treated as a variable by Prolog. However, due to the problem mentioned before, we must use a lowercase letter in order to retrieve its name. The solution here is to use a pair of names at each occurence of a value.

For example:

... V12, v12 ...

2) Internal representation

The internal representation is defined in terms of BNF-like rules.

However, these rules are a bit informal:
- Non-terminals begin with the symbol 'P_'.
- Repetitions of zero or more times are surrounde by the
brackets '{' and '}'.
- Square brackets represents terminal symbols.

```
P_internal_representation := P_flat_specification
                             P_canonical_type
                             P_renaming_function
/* Behaviour Description */
P_flat_specification := { P_process . }
P_process := proc(P_procid , P_gateids , P_valuedecls , P_behexp)
P_behexp := let(P_valueqns , P_behexp)
          | distchoice([ [P_valdecl] , P_linenum ] , P_behexp)
          | distchoice( P_gateid , P_gateids , P_behexp)
          | distpar( P_gateid , P_gateids , P_parop , P_gateids ,
                     P_behexp)
          | enable(P_behexp, P_valuedecls , P_behexp)
          | disable(P_behexp , P_behexp)
          | parcomp(P_parop , P_behexp , P_gateids , P_behexp)
          | choice([ P_behexp {, P_behexp} ])
          | guard(P_simpleqn , P_behexp)
          | hiding(P_gateids , P_behexp)
          | seq(P_event , P_behexp)
          | instance(P_processid , P_gateids , P_expressions)
          | nested(P_behexp)
          | exit([ exit(P_linenum) ,
               [ P_experiment_accept {,P_experiment_accept}],
               [] ])
          | stop(P_linenum)
P_event := i(P_linenum)
         | [ P_gateid(P_linenum) ,
             [ P_experiment {, P_experiment} ], P_guard ]
P_experiment := P_experiment_normal
          | P_experiment_accept
P_experiment_normal := [# , P_expression , P_sortid]
P_experiment_accept := [$ , P_valdecl]
P_valueqns := [P_valueqn {, P_valueqn}]
P_valueqn := [P_valdecl , P_expression]
P_valuedecls := [[P_valdecl] {, [P_valdecl] }]
P_valdecl := P_Valid , P_valid , P_sortid
P_guard := []
         | P_simpleqn
P_simpleqn := [ P_expression , P_expression ]
          | [ P_expression ]
P_expressions := [ P_expression {, P_expression} ]
P_expression := P_opnid
```

```
                | P_opnid(P_expression {, P_expression})
                | val( P_Valid , P_valid )
P_gateids := [P_gateid {, P_gateid}]
P_parop := interleave
          | maxsync
          | selected
P_linenum := [ P_number_id ] /* taken from the abstract tree */
/* Abstract Data Types Description */
P_canonical_type := { P_sort . } { P_equation . }
P_sort := sort P_sortid => P_expression
P_equation := [ [P_premis {, P_premis }] ] =-=> P_simpleqn2
P_premis := P_expression2 =--= P_expression2
P_simpleqn2 := P_expression2 ==> P_expression2
P_expression2 := P_expression2 {, P_expression2 }
P_expression2 := P_opnid
                 | P_opnid(P_expression2)
                 | P_Valid
/* Renaming Function Description */
P_renaming_function := { renm(P_renaming) . }
P_renaming := P_specifid , Atom , [ ]
          | P_id , Atom , [P_nesting]
P_id := P_typeid
          | P_processid
          | P_sortid
          | P_opnid
          | P_gateid
          | P_valid
P_nesting := P_specifid
               | P_processid
               | P_typeid
/* Identifiers Description */
P_procid := P_processid
             | P_specifid
P_specifid := sp Index
P_processid := p Index
P_typeid := t Index
P_sortid := s Index
P_opnid := o Index
          | true
P_gateid := g Index
             | exit
P_valid := v Index
P_Valid := V Index
```

Note: Index is the unique index associated with each identifier in the specification.

Appendix F: Inference Rules Implementation

```
        /* Axioms */
        branch(seq(i(LN),B),i(explicit),B,LN,Pr,Pr).
        branch(seq([A|L],B),[G|L],B,LN,Pr,Pr) :- A =.. [G,LN].
        branch(exit([exit(LN),V,[]|T]),[exit,V,[]|T],stop,LN,Pr,Pr).
        branch(distchoice([[V1,V2,S],LN],B),[dummy,[V1,V2,S],[]],B,
         LN,Pr1,Pr1).
        /* Inference Rules */
        branch(let(C,B),Action,BR,LN,Pr1,Pr2):-
            asglet(C,B,BL), /* Instantiate values of 'let' in B */
            branch(BL,Action,BR,LN,Pr1,Pr2).
        branch(guard([PD],B),A2,BR,LN,Pr1,Pr2):-
            tree, /* in the case of symbolic tree, */
                /* guards are assumed true if */
                /* they are not evaluated to false */
        eval(PD,P),P \== false,
        branch(B,A,BR,LN,Pr1,Pr2),
        branch(guard([PD],B),A,BR,LN,Pr1,Pr2):-
            \+tree, /* for normal simulation,  */
                /* guards should be evaluated to true */
        eval(PD,true),
        branch(B,A,BR,LN,Pr1,Pr2).
        branch(choice(BL),A,BR,LN,Pr1,Pr2):-
        member(B1,BL), /* for every B1 in BL */
        branch(B1,A,BR,LN,Pr1,Pr2).
        branch(hiding(HD,B),Action2,hiding(HD,BR),LN,Pr1,Pr2):-
            branch(B,Action,BR,LN,Pr1,Pr2),
            hide_or_not(Action,HD,Action2).
        branch(nested(B),A,BR,LN,Pr1,Pr2):-
            branch(B,A,BR,LN,Pr1,Pr2).
        branch(parcomp(CT,B1,CG,B2),A,BF,LN,Pr1,Pr2):- allsolutions(B1,S1), /* all actions of
B1 */
        allsolutions(B2,S2),!, /* all actions of B2 */
            /* check synchronization type */
            check_parallel([B1,S1],[B2,S2],CT,CG,A,BF,LN,Pr1,Pr2).
        branch(enable(B1,PR,B2),A2,BR2,LN,Pr1,Pr2):-
        branch(B1,Action,BR,LN,Pr1,Pr2),
            /* check if successful termination */
            enable_exit(PR,Action,A2,BR,B2,BR2).
        branch(disable(B1,B2),Action,BR2,LN,Pr1,Pr2):-
        branch(B1,Action,BR1,LN,Pr1,Pr2),
            /* check if successful termination */
        disable_exit(Action,BR1,B2,BR2).
branch(disable(B1,B2),Action,BR,LN,Pr1,Pr2):-
```

```
branch(B2,Action,BR,LN,Pr1,Pr2).
branch(relabel(B,RG),Action,relabel(BR,RG),LN,Pr1,Pr2):-
branch(B,Action1,BR,LN,Pr1,Pr2),
        /* check if relabelling should occur */
relabels(Action1,Action,RG).
branch(distchoice(G,G2,B),A,BR,LN,Pr1,Pr2):-
        member(G21,G2), /* for every gate in G2 */
        branch(relabel(B,[[G],[G21]]),A,BR,LN,Pr1,Pr2).
branch(distpar(G,G2,OP,Gs,B),A,BR,LN,Pr1,Pr2):-
        length(G2,COPIES),
        make(B,COPIES,Bs), /* make COPIES copies of B */
        simplify_B(G,G2,OP,Gs,Bs,B2), /* simplify them */
        branch(B2,A,BR,LN,Pr1,Pr2).
branch(instance(A,G,V),Action,BR,LN,Pr1,[instance(A,G,V)|Pr2]):- evalparm(V,VR),/*evaluate
actual parameters */
proc(A,OG,VR,B), /* substitute formal parametrs */
                        /* by actual parameters */
branch(relabel(B,[OG,G]),Action,BR,LN,Pr1,Pr2).
```

References

[BB]B. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", North-Holland, Computer Networks and ISDN Systems 14, pp.25-59, 1987.

[BJ]D. Brand, W.H. Joyner Jr., "Verification of Protocols Using Symbolic Execution", Computer Networks 2,4/5, pp.351-360.

[BRI]E. Brinksma, "A Tutorial on LOTOS", in: B. Sarikaya and G.v. Bochmann (eds) Protocol Specification, Testing, and Verification VI, North-Holland, pp.171-194, IFIP, 1986.

[EFH]H. Ehrig, W. Fey, H. Hanson, "ACT ONE: an algebraic specification language with two levels of semantics", Bericht-Nr. 83-03, Technical University of Berlin, 1983.

[EIJ]P. Van Eijk, "Software Tools for the Specification Language LOTOS", Ph.D. Thesis, University of Twente, January 1988.

[FEH]M.C. Fehri, "A System for Validating and Executing LOTOS Data Abstractions (SVELDA)", MCS Thesis, University of Ottawa, 1987.

[GIL]D. Gilbert, "Executable LOTOS: Using PARLOG to implement an FDT", in: H. Rudin and C.H. West (eds) Protocol Specification, Testing, and Verification VII, North-Holland, pp.281-294, IFIP, 1987.

[Hoare]C.A.R. Hoare, "Communicating Sequential Processes", Prentice-Hall International, London, 1985.

[JDM]Jan de Meer, "Tutorial on LOTOS Data Types". Hahn -Meitner-Institut Berlin GmnH,1986.

[ISO1]International Organization for Standardization, Information Processing Systems, Open Systems Interconnection, "LOTOS - A Formal Description Technique Based on Temporal Ordering of Observational Behaviour" (ISO DIS 8807), 1987.

[ISO2]International Organization for Standardization, "Formal Description of ISO 8072 in LOTOS", (ISO/TC 97/SC 6/N 317), 1987.

[OBA1]A. Obaid, "Applications of the inference rules of CCS and LOTOS", University of Ottawa, Department of Computer Science, Technical Report 85-14, 1985.

[OBA2]A. Obaid, "SINAPS: A simulator of communicating systems", University of Ottawa, Department of Computer Science, Technical Report 86-14,1986.

[PAP] G. Pappalardo, "Experiences with a Verification and Simulation Tool for Behavioural Language", in: H. Rudin

and C.H. West (eds) Protocol Specification, Testing, and Verification VII, North-Holland, pp.251-264, IFIP, 1987.

[KOR] D.G Kourie, "The Design and Use of a Prolog Trace Generator for CSP", University of Pretoria, Software-Practice and Experience, Vol. 17(7), pp.423-438, 1987.

[LG] L. Logrippo, R. Guillemot, "Derivation of Useful Execution Trees From LOTOS Specifications by Using an Interpreter", University of Ottawa, Department of Computer Science, Technichal Report 88-13, 1988.

[LGH] L. Logrippo, R. Guillemot, M. Haj-Hussein, "Executing Large LOTOS Specifications", University of Ottawa, Department of Computer Science, Technichal Report 88 -03, 1988.

[LO] L. Logrippo, A. Obaid, "Outils Logiciels pour le Langage LOTOS", University of Ottawa, Department of Computer Science, Technichal Report 88-16, 1988.

[LOBF] L. Logrippo, A. Obaid, J.P. Briand, M.C. Fehri, "An Interpreter for LOTOS: A Specification Language for Distributed Systems", To appear in Software - Practice & Experience.

[MIL] R. Milner, "Calculus of Communicating Systems", Vol. 92, Springer-Verlag 1980.

[URS] H. Ural, R. Short, "An Interactive Test Sequence Generator", in: Communications Architectures and Protocols, SIGCOMM ’86 Symposium, pp.241-250.

[VIS] C. Vissers, "Architecture and Specification Style in Formal Description of Distributed Systems", in: S. Aggarwal, K. Sabnani (eds) Protocol Specification, Testing, and Verification VIII, IFIP, 1988.