Abstract

LOTOS (Language Of Temporal Ordering Specification) is one of the currently available Formal Description Techniques developed for the specification of OSI. It is based on the temporal relation among the interactions that constitute the externally observable behaviour of a system. On the other hand, PROMELA is a language used to describe both validation and specification models. The topic of this thesis is the design and implementation of a translator from LOTOS specifications into PROMELA validation models. LOTOS specifications are first syntactically and semantically checked and then they transformed into a network of extended finite state machines. The automata are then translated into PROMELA where they synchronise according to a multi rendezvous algorithm. Unfortunately, it is proved that due to the complexity of LOTOS multi synchronisation the number of states produced is too large to be analysed.

Translating LOTOS Specifications into PROMELA

By

Nicholaos C. Petalidis

Submitted for the Degree of

Master of Philosophy

at Heriot-Watt University

on Completion of Research in the

Department of Computing and Electrical Engineering

October 1994.

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Nicholaos C. Petalidis (Candidate)

Dr. Peter J.B King (Supervisor)

Date

Contents

List of Figures

Acknowledgements

I am most grateful to my supervisor Dr. Peter J.B. King for all the guidance and valuable advice that he has given to me throughout my graduate studies.

I also like to thank my friends A. Kaloxylos and C. Farmakis for their help.

Special thanks to my parents Christos and Irene that supported me throughout my undergraduate and graduate studies.

Chapter 1

Introduction

1.1 Prologue

The evolution of computer networks gave rise to the problem of specifying and verifying "correct" protocols for the error free communication between computer systems.

The International Organisation for Standardisation (ISO) in an attempt to find solutions to this problem has worked towards the creation of a set of formal languages that enable the unambiguous specification of protocols and services.

On the other hand, a protocol must not only be "unambiguously" specified, but it must be guaranteed to work under all possible circumstances. Various techniques for the analysis of the behaviour of protocols have been proposed and implemented and a number of different *validators* are now in existence.

1.2 Organisation of the Thesis

In the following sections, a closer look at formal description techniques (FDTs) will be made. A discussion on the notion of verification and its importance in the development of protocols is presented and finally, an outline of the motives for this project as well as a review of projects in related areas is made.

Chapters 2, 3, 4 give an overview of the LOTOS specification language, the PROMELA verification language and describe the features of the SPIN validator.

In Chapter 5 a detailed description of the translator is made. The process of producing extended finite state machines from basic LOTOS specifications is explained in detail. Chapter 6 explains the protocol followed for the synchronisation of the finite machines.

A discussion on the final result is made in Chapter 7. The difficulties of obtaining an extended analyser for basic LOTOS specifications are presented. We also present a different rendezvous algorithm, that uses a minimum number of message transfers. Unfortunately, it also fails to keep the state space produced for the LOTOS specifications small.

Chapter 8 explains why the data part has not been incorporated in the translator.

In the last chapter, Chapter 9, the reader can find guidelines and ideas for extending this work.

1.3 Overview of FDTs

Formal Description Techniques (FDTs) are methods of defining the behaviour of an (information processing) system in a language with a formal syntax and semantics, instead of a natural language [ISO89a].

FDTs give us the ability to specify systems in a consistent way avoiding any possible misunderstanding that may lead to incompatible implementations. Moreover, FDTs allow for the analysis of protocols to be made on a mathematical basis.

The three FDTs in common use today are LOTOS [ISO89a], ESTELLE [ISO89b] and SDL [CCI89]. The first two techniques, (LOTOS and ESTELLE) were developed particularly for the specification of OSI protocols but nowadays they are used anywhere where the specification of a parallel protocol is needed. The process parts of SDL and ESTELLE are based on Extended Finite State Machines

(EFSM), a model that was quite famous in protocol specification methods from the early years. Although EFSM-based languages are easily implemented they lack of the abstractness needed to write machine - independent protocols.

LOTOS came to fill this gap. Based on a whole different philosophy it allows one to produce specifications that are not dependent on a particular system. Unfortunately, it is due to this abstractness that LOTOS is the hardest to implement of the formal description techniques.

The data parts of both LOTOS and SDL are based on abstract data types and more precisely on the language ACT-ONE [EM85]. This is something that gave both languages extra strength. Programmers can define their own data and operations and they can generalise their functions wherever possible. On the other hand, this is proving to be cumbersome for the people trying to derive running prototypes from these languages. The more abstract a language is, the more difficult it is to derive a concrete implementation from it.

Finally, the data part of ESTELLE is more concrete, having a lot similarities with the PASCAL programming language.

Apart from the three description techniques mentioned above a number of different languages exist for the formal specification of systems, although they are not considered to be FDTs. Examples of such languages are the specification languages Z [MS92], VHDL [Per94], and RAISE [Geo91].

1.4 Protocol verification

Verification became an important part of protocol design as computer systems became faster and more complex. The basic idea behind verification is that a method is needed to ensure that a protocol does not contain any logic contradictions and it does what it is supposed to do. Although the last sentence may seem a little bit naive, (why should anyone write a protocol that does not do the job it was created for) it must be realised that when we describe complex systems it is very difficult to take under consideration all the possible states they may fall in.

Apart from that, the term *validation of protocols* is also used in a broader sense and its interpretation depends on the user's interests and objectives. Thus, the objective could be to verify that a specified service satisfies the user requirements. Another objective could be to prove that a given implementation of a protocol correctly conforms to its specification [Peh90].

In this context the term is used to refer to some kind of proof that a protocol satisfies a given set of criteria, or in other words that the protocol actually provides the service we want.

Thus, to verify a protocol we need two things: A specification of the protocol itself and a specification of the properties the protocol must respect.

So, in order for the verification procedure to understand and analyse a specification, both the protocol and its properties have to be written in some predefined notation.

There is a number of different methods that can be used for the specification of protocols and their properties. Common approaches are the use of state machines, petri nets or state-charts. Properties are usually specified in terms of assertions associated with a transition system [Peh90].

Furthermore, the verification procedure must rely on some predefined algorithms for proving that the protocol satisfies its properties [GG93]. These algorithms either try to exhaustively explore every possible execution sequence of the protocol (*reachability analysis*) or they trace only those sequences that are more likely to produce an error (*state space explosion*). In Chapter 4 a detailed description of these techniques can be found.

In addition, it is possible to prove that the protocol under consideration is actually *equivalent* to another one that it is either simpler or known to be correct. Similarly a protocol can be validated if its specification is proved to be *equivalent* to the specification of the service it should provide. In general, two specifications are said to be equivalent when they can be used interchangeably. There are a

number of different equivalences in use today. Some of them require the specifications to be almost identical while others have more relaxed laws. The reader can find more information on equivalence relations in references [MS88, Hen88].

To help the process of verification a number of tools have been developed that aim either to fully automate the validation process or to help in cases where a fully automated method can not be followed.

Examples of such tools [JT93] are *state space analysers* which try to follow every possible execution sequence of a specification and *simulators* which trace a particular execution sequence. Other tools can be used to prove the equivalence of two specifications (*equivalence checkers*) or to derive test cases that can be used to check if an implementation conforms to its specification (*test generators*).

1.5 A Lotos to Promela Translator: Why?

The previous discussion made clear that in order to apply an automated method (e.g. reachability analysis) for verifying a protocol we need to express the protocol and its properties into a form acceptable by some validation tool.

These days the number of protocols expressed in the specification language LOTOS is continually increasing. However, LOTOS is not intended for validation purposes and this means that it is necessary to find a way for verifying LOTOS protocols. One method, of course, is to write validation tools that can directly understand and manipulate LOTOS inputs. Projects that are oriented towards this direction are presented in the next section.

Another method is to use existing tools and translate LOTOS specifications in a form that these tools understand. This is the method followed here.

The validation tool used is SPIN[1]. SPIN is a very powerful reachability analysis tool. It uses the most up-to-date validation algorithms and it allows for reachability analysis, state space explosion and random simulation. Moreover, its

[1]As of March '95 a new version of SPIN(v. 2.0) is available. However, the version of SPIN used in this project is an earlier one (v. 1.5.14).

input language, PROMELA, can be used to express any correctness criteria and assertions that the user feels are necessary to be proven.

A tool that translates LOTOS specifications into PROMELA will provide protocol designers with the flexibility and abstractness they need for their specifications (using LOTOS) enhanced with the guarantee of the protocol's correctness (using SPIN).

In addition, a translator of LOTOS combined with the analogous translator of ESTELLE [Kal95] can be used to compare or even combine specifications written in these languages.

1.6 Related Work

Several attempts have been made to obtain running implementations from LOTOS specifications. In this section a description of the compilers and simulators for LOTOS currently available will be made.

- The SEDOS project

  The SEDOS project, begun in 1984 and finished in 1987, had the primary aim of further developing LOTOS and ESTELLE formal description techniques. SEDOS produced a number of tools and methods for simulating, verifying and testing LOTOS specifications [vE89, Tre89]. A great deal of the work currently going on in LOTOS is based on the results of the SEDOS project.

- TOPO

  The Technical University of Madrid has created TOPO, a set of tools that allows the compilation of both the data type and control part of LOTOS and the derivation of code written either in C or in ADA. For the translation of the data part a rewrite system is generated while a network of finite state machines synchronised by a central controller comes to serve as a model for

the control part. The tool-set also enables the use of executable comments that try to minimise the abstractness of a specification. TOPO covers full LOTOS [AMnDSA93, Tur93].

- LOTOS interpreter of the University of Ottawa

  A number of research projects concerning LOTOS have been carried out by the University of Ottawa. The main tool created there is an interpreter consisting of two parts: one that handles the data types of LOTOS (the SVELDA interpreter) and one that handles the control part of the language (the ISLA interpreter). The tool validates and evaluates value expressions and produces prototypes from LOTOS specifications and it is written in the PROLOG programming language [GL93, Tur93].

- CADP (Caesar/Aldebaran Distribution Package)

  The CADP tool-box is dedicated to the efficient compilation, simulation, formal verification, and testing of descriptions of LOTOS specifications. The tool-box contains several closely interconnected components: ALDEBARAN, CAESAR, CAESAR.ADT, and OPEN/CAESAR.

  ALDEBARAN

  ALDEBARAN is a tool for verifying communicating systems, represented by labelled transition systems (LTS), i.e. finite state machines, the transitions of which are labelled by action names.

  CAESAR

  CAESAR is a compiler which translates LOTOS descriptions into LTSs. It interfaces a number of verification tools for LTSs and temporal logic evaluators.

  CAESAR.ADT

  CAESAR.ADT is a compiler that translates the data part of LOTOS specifications into libraries of C types and functions.

  OPEN/CAESAR

OPEN/CAESAR is an extensible environment based upon CAESAR and CAESAR.ADT. It allows user-defined programs for simulation, execution, verification (partial, on-the-fly, etc.), and test generation to be developed in a simple and modular way.

The tool accepts full LOTOS with the exception of specifications that contain dynamically created processes [CFGM+92, Tur93].

- LOLA

  LOLA is a transformational and state exploration tool used for testing, simulation and debugging purposes. LOLA can produce finite state machines, extended finite state machines or behaviour trees. It has options for simulating a LOTOS specification or evaluating data expressions. LOLA covers full LOTOS [QPF90, Tur93].

- SMILE

  SMILE is a symbolic evaluation tool for LOTOS which can analyse the data part and produce a rewrite system. It has functions for the execution and debugging of a specification and can produce extended finite state machines that are equivalent (strong bisimulation equivalence) with the specification. SMILE covers full LOTOS but it can only execute finite specifications [EW93, Tur93].

- Other tools

  A number of other tools exist, that either translate LOTOS sources into another language, such as the LOTOS to PARLOG translator [Gil88] or produce executable code not only for LOTOS but for other formal description techniques as well ([BBD+92]).

Nowadays, research efforts are directed towards the translation of LOTOS specifications into a network of extended finite state machines that co-operate according to some multi-rendezvous algorithm([Kar92, VSC90]). Labelled transition systems have been proven to produce inefficient implementations as they are not able to handle non regular processes correctly.

Finally, it should be mentioned that efforts are made to formally define and standardise extended versions of LOTOS that can handle real time [LLdF+94]. Time extended LOTOS is extremely useful for applications containing real time, such as video, audio and industrial applications and the production of tools that can handle the extended versions will be of crucial importance.

1.7 Epilogue

In this chapter the concept of specification and verification of protocols was introduced. A presentation of the three formal languages in use today was made and the need for a tool that will translate specifications written in these languages into another format was explained. The chapter closed with a review of projects and tools related with this subject. A detailed description of this project can be found in the subsequent chapters.

Chapter 2

An overview of LOTOS

2.1 Prologue

LOTOS (Language Of Temporal Ordering Specification) is a formal description technique standardised for the OSI (Open Systems Interconnection) services and protocols. LOTOS specifications describe distributed systems by defining the temporal relations among the interactions that represent the system's externally observable behaviour [BB87].

LOTOS specifications consist of two components:

1. A control component based on Milner's Calculus of Communicating Systems (CCS [Mil80, Mil89]) and Hoare's Communication Sequential Processes (CSP [Hoa85]), which deals with the description of process behaviours and interactions, and

2. A data component based on the formal theory of algebraic abstract types ACT-ONE [EM85], that describes the data structures and value expressions.

Due to the mutual independence of the two components of LOTOS it is possible to write specifications which do not make use of data values; they constitute the subset of LOTOS called basic (or pure) LOTOS. Actually, this part is the CCS/CSP component of LOTOS.

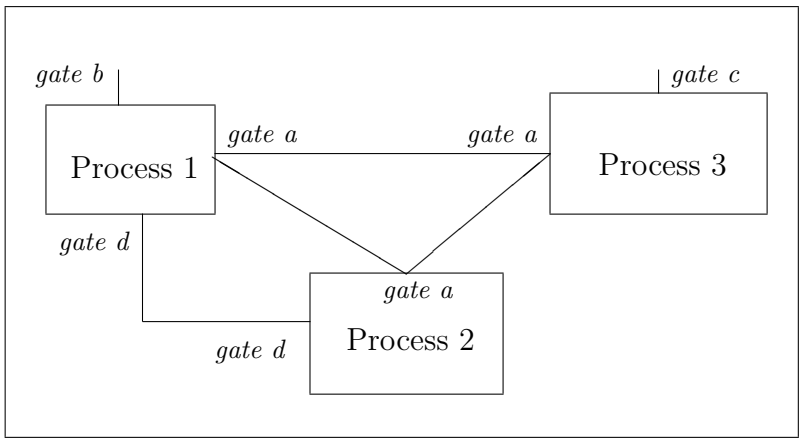

Figure 1: Processes interacting with each other

2.2 General Principles

In LOTOS a system is specified as a process possibly made up of several interacting subprocesses, each of which is a process too and may itself be divided into subprocesses. We can also assume that the observer of the system is a process ready to observe any observable action.

Interaction among processes takes place at communication points called *gates*. Thus, every process is characterised by a number of gates. Every observable action in which a process can participate is expressed by an action denotation composed of a gate identifier (where the action may occur) and, optionally, a data exchange part. Actions in LOTOS are *atomic*, i.e. it is considered that they do not consume time. When a process is ready for an interaction at some gate we say that it is ready to *offer* an observable action at that gate.

In general we can think of processes in LOTOS as being black boxes communicating with each other and the environment by a number of predefined gates, like in Figure 1. There we have three processes that interact with the environment (via gates b, c) and with each other (via gate d or gate a).

2.3 Basic Lotos

The control component of LOTOS deals with the description of process behaviours and interactions.

The syntax of a process definition looks like:

```
process <proc-name> [list-of-gates]
        <behaviour-expression>
where process <proc-name> [list-of-gates]
              <behaviour-expression>
      endproc
endproc
```

A behaviour-expression relates the order in which events may occur and is built up by applying an operator to other behaviour expressions. The precedence of each operator is:

action prefix > choice > parallel composition > disabling > enabling > hiding

The semantics of each of the operators are described below:

2.3.1 Inaction: stop

This operator indicates the completely inactive process. It does not offer anything to the environment and it does not have any internal actions to perform. **Stop** can be thought as being equivalent to *deadlock* if no other possibilities exist.

2.3.2 Action prefix

Syntax: **i**;*B or g*;*B*

where *B* is a behaviour expression, *g* a name of a gate and **i** a special gate-name that denotes the unobservable internal action.

This operator expresses sequential composition of actions. In general it means that an event must occur at the specified gate in order for the process to proceed. As soon as the event occurs, the process behaves according to expression B.

2.3.3 Choice

Syntax: $B_1[]B_2$

The *choice* operator offers a choice between two behaviour expressions $B_1,\ B_2$. The choice is usually resolved by the environment. If both choices are equally possible the result is a non-deterministic one. It should be mentioned here that once an action is chosen from one of the components, the other component disappears from the resulting expression.

This means, that if we had an expression:

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

and the first action chosen was a then the only possible action afterwards would be b. The right part of the expression was dropped as soon as the selection was made.

2.3.4 Parallel Composition

Syntax: $B_1|[list\ of\ gates]|B_2$

This operator denotes the case where B_1 and B_2 are executed independently but synchronising at the gates mentioned in the list *'list of gates'*.

If the *'list of gates'* is the empty set then the two processes will be executed in an interleaved way. We denote this special case by using the ||| operator. If, on the other hand, we want to achieve full synchronisation between two processes, i.e both of them must synchronise in every gate, then we use just the || operator leaving unspecified the list *'list of gates'*. In every case the processes must synchronise upon exiting.

It is also possible that in some cases more than two processes may synchronise at the same gate. For example, the expression:

$$(a; b; exit|[a]|a; d; exit)|[a]|a; c; exit$$

causes gate a to be the synchronisation point of three different processes. All the three processes must offer an event at the same time. Otherwise the processes will be blocked. This means that if we had something like

$$(a; b; exit|[a]|a; d; exit)|[a]]b; a; exit$$

then we would be dealing with a *deadlock* situation.

2.3.5 Hiding

Syntax: **hide** *list of gates* **in** B

A **hide** operator hides the gates listed in the *'list of gates'* from external processes. If an action is not hidden, with respect to the environment, cooperation of the environment is required. By hiding an action we are able to make the synchronisation of two processes invisible and disable the use of the named gates by any other process.

2.3.6 Successful termination: exit

The keyword **exit** indicates that a process has successfully terminated. This means that execution can continue to other processes as mentioned in the next section.

2.3.7 Enabling

Syntax: $B_1 >> B_2$

The LOTOS enable operator has a similar function as the action prefix operator. It denotes that process B_2 must start execution just after the successful termination of process B_1. It should be noted that B_1 must terminate successfully (i.e through the **exit** operator) in order for B_2 to start executing.

2.3.8 Disabling

Syntax: $B_1[> B_2$

The LOTOS disable operator [> models an interruption of a process by another process. It means that in any point of the execution of B_1, process B_2 can interrupt B_1 and take over control. Once B_2 starts executing the actions of the other process are no longer possible. On the other hand, if B_1 terminates before B_2 takes over control then B_2 disappears.

2.3.9 Process Instantiation

Syntax: $B{=}P[g_1, ..., g_n]$

By process instantiation, the behaviour of B becomes the behaviour of the process which is defined as P. The formal gates appearing in the definition of process P are substituted by the actual ones appearing in the expression above. We can define infinite behaviours in LOTOS by using recursive process instantiations.

2.4 Data Types

LOTOS data type part is based on the specification language for abstract types ACT-ONE. An *abstract data type* enables the user to specify his own data types and the corresponding operations on them, just as he is able to specify his own functions by the use of procedure declarations in conventional programming languages. The actions one must take, to define his own data types, are described below.

2.4.1 Signature

The first step in specifying a data type is to define its *signature*. Let's take a closer look at the example presented in Figure 2.

The *signature* of a data type defines its *sort* (type) and the operations permitted on this sort. *Sorts* can be viewed as named sets of elements. In our example

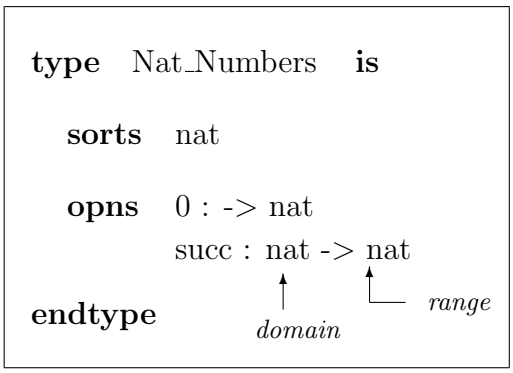

Figure 2: Example of a signature

we have defined a new type *Nat_numbers* that is of sort *nat* and we have allowed two operations: The *succ* operation is performed on a data type of sort *nat* (the domain of the operation) and the result is another data type of the same sort (the range of the operation). The other operation is a nullary operation (*constant*) that results to a type of sort *nat*. *Operations* are functions that map a number of elements taken from a particular set (the *domain*) into a single element of another set, the *range*. We can construct all the elements of a sort if we know just one element of that sort by repeatedly applying a nullary operation to that element.

For example, we can view *0* as a *term* of type *Nat_numbers*. A *term* is the result of applying an operation to other terms. For example some terms of the type *Nat_numbers* are the following:

0, succ(0), succ(succ(0)), ...

2.4.2 Equations

Equations provide us with means to prove the equivalence of two terms that belong to the same sort. *Equations* do not define new terms. They make our life easier by helping us to use simple terms that are equivalent with other more complex ones.

For example, if we add the following piece of code to the specification of the

signature of type *Nat_Numbers* we will define an operation for addition.

eqns

forall x,y: nat

ofsort nat

x + 0 = x

x + succ(y)=succ(x+y)

Although we did not add any new terms to the set of *Nat_Numbers* we expressed a convenient method for defining members of this set. Instead of using the term $succ(succ(0)) + succ(0)$ we can now equivalently use the term $succ(succ(succ(0)))$.

Because this is a brief description of the language we will not expand further. We shall only outline some of the other features of LOTOS data types.

2.4.3 Extension

After we have defined the signature of a type we can *extend* it by providing it with new operations, sorts and equations. This way the number of expressions derived from the data type is greatly enlarged.

2.4.4 Combining

Types can also be combined. Complex types can be easily constructed by using simpler ones. By combining sorts, operations and equations we obtain richer signatures that define more complex data structures.

2.4.5 Renaming

It is often the case that some data types are similar. Therefore, the renaming facility of LOTOS is used to generate a new independent type by changing the

names of its components. Renaming is used every time we want to define a new type that has been used before, probably with another name.

2.4.6 Parameterised Data Types

LOTOS data types are flexible enough to permit the *reuse* of *signatures* and *operations*. This is possible through the use of *parameterisation*. *Parameterised data types* look a lot like functions that can be called with certain parameters and perform a predefined set of operations on that parameters. They consist of a formal body with formal sorts and operations and a main body with its own sorts and operations much like a function's body. Each time a parameterised data type is called, its formal parameters are replaced by the parameters provided by the caller. A parameterised data type looks like:

```
type ... is
      formalsorts ...
      formalopns ...
      sorts ...
      opns ...
      eqns ....
endtype
```

2.5 Full LOTOS

Full LOTOS combines the power of basic LOTOS with the characteristics of abstract data types. In full LOTOS, it is possible to describe process synchronisation involving the exchange of data values. Thus, interprocess communication is available.

A specification written in full LOTOS is of the form:

specification spec_name[gate list](parameter list):functionality

type definitions

behaviour

behaviour expression

where

type definitions

process definitions

endspec

A process definition in full LOTOS has the form:

process proc_name[gate list](parameter list):functionality:=

behaviour expression

where

type definitions

process definitions

endproc

2.5.1 Action prefix

In full LOTOS an action is formed of three components:

1. A gate that denotes the place where the event is going to happen
2. A list of events.
3. An optional predicate

Consider the LOTOS expression presented in Figure 3. In this example we say that an event will occur at gate g. The variable *Get* will take an arbitrary value from the set of *Nat_Numbers* such that *Get*>0. We say that in this case we have a *variable declaration* and we can think that our process asks for a value

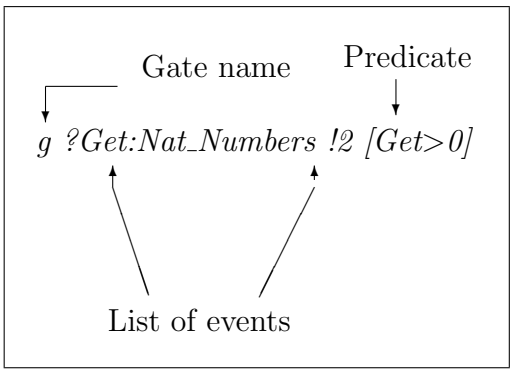

Figure 3: Actions in full LOTOS

to be offered (by some other process). After that, later instances of this variable expression will be replaced by this offered value. The other event is a *value declaration* and we can think of it as a value offer. In general we can think that the **?** symbol indicates input, whereas the **!** symbol indicates output.

2.5.2 Parallel Composition in Full LOTOS

The semantics of parallel composition do not change. The only constraints added are those that have to do with value offers.

Two processes can proceed in parallel if they offer the same value (*value matching*), or wait for an input of the same sort. In the latter, the same (random) value is assigned to both variables (*value generation*). If one process offers a value, while the other one is waiting for a value of that sort, then we actually have a *message passing* situation where one process takes the value offered by the other.

The reader can refer to other papers (e.g [ISO89a]) for a detailed description of the synchronisation conditions.

2.5.3 Successful termination with parameters

This is denoted by **exit**(E_1,...,E_n). The process that exits through this operator offers the values E_1,...,E_n. In order for two processes to be synchronised they

have to offer at the end the same set of values (in LOTOS terminology we say that they must have the same **functionality**).

2.5.4 Guarded behaviour

A behaviour expression may also be preceded by some guard that must be true in order for the former to be enabled. For example, the expression:

$$[X < 2] -> g!X;\textbf{stop}$$

will enable the offer of X only if its value is smaller than 2.

Guarded behaviours are extensively used in order to avoid specifications that lead to infinite execution trees.

2.5.5 Sequential composition with value passing

In full LOTOS in order to connect two processes with the >> operator we have to use the following schema:

$$B_1 >> \textbf{accept } X_1 : S_1, ..., X_n : S_n \textbf{ in } B_2$$

B_1's **functionality** must match with the sorts listed after the **accept** operator. If B_1 exits then the next behaviour expression B_2 becomes active and the variables X_1 to X_n in B_2 are substituted by the value results of B_1 (i.e the **exit** parameters).

2.5.6 Generalised choice

LOTOS provides us with a convenient way for expressing an infinite number of alternatives.The expression:

choice *x:Nat_Numbers[]a;B(x)*

declares that x may be any of the natural numbers. After a choice from the set of natural numbers, the value of *x* will be placed in the expression *B(x)*. The choice can be either non deterministic or it can be determined by the environment. We

can denote in a similar way a generalised choice over a set of gates. The expression has the following syntax:

$$\textbf{choice } g \textbf{ in } [g_1, ..., g_n][]B$$

2.5.7 Parametric processes

In full LOTOS we can parameterise process definitions by using variable declarations together with the list of formal gates. The syntax looks like the one in basic LOTOS:

$$\textbf{process } \textit{process\_name } [\textit{gate\_list}](x_1 : s_1, ..., x_n : s_n) \textit{ :functionality} := ...$$
$$\textbf{endproc}$$

Whenever an expression of the form:

$$\textit{process\_name } [\textit{actual gate parameters}](v_1, ...v_n)$$

is met, process *process_name* is instantiated and the formal parameters $x_1, ..., x_n$ are replaced by the actual ones: $v_1, ..., v_n$.

2.5.8 Local Definitions

We can replace a value identifier (i.e. a variable name) with a value expression by using the **let** construct of full LOTOS. The statement:

$$\textbf{let } x_1 : Sort_1 = v_1, ..., x_n : Sort_n = v_n \textbf{ in}$$

will replace all occurrences of

$$x_1, ..., x_n$$

by the respective value expressions

$$v_1, ..., v_n.$$

2.5.9 Par Expression

Similar to the generalised choice expression is the **par** expression. Its syntax is:

$$\textbf{par } g \textbf{ in } [g_1, ...g_n] \ op \ B$$

where *op* is any parallel operator. The expression is equivalent to

$$B[g/g_1] \ op \ ... \ op \ B[g/g_n]$$

where g/g_i means: *replace* g *with* $g_i, i = 1, ..., n$

2.6 Epilogue

The above discussion about LOTOS (and especially the one referred to full LOTOS) is meant to be only a brief overview. The reader who wants to become familiar with the language is advised to read other tutorials, for example [JT93], [BB87] or [LFH92].

What the reader must have in mind, is that LOTOS gives us the ability to specify a system in various levels, from the highly abstract to the more detailed one. Through the years various techniques have been developed for specifying systems. Some are meant to ease the work of the implementer, others that of the specifier. Therefore, although LOTOS is an executable language, not all specifications may be able to produce running code. The problems of generating running code from LOTOS programs are discussed below.

Chapter 3

An overview of PROMELA

3.1 Prologue

PROMELA is a language for writing validation models. Its syntax is in many aspects a lot like C. The following is meant to be only a brief overview. The reader who wants a tutorial in PROMELA should consult reference [JH91].

3.2 Statements

All the statements in PROMELA except the assignments and declaration of variables are conditionally executed i.e. statements are executed only when they can. For example, the boolean condition

$$a{==}b$$

is executed only when both a and b have the same value. Otherwise the statement is blocked. This is how PROMELA achieves synchronisation.

3.3 Variables, Data types

PROMELA supports the following data types:

bit, bool, byte, short, int, chan.

The first five types are basic data types, supported by almost any programming language. The data type *chan* is used to model message channels, enabling the transfer of data from one process to another.

PROMELA also supports the use of one dimensional arrays for all of the above data types.

3.4 Procedures

Procedures in PROMELA can be recursive, can accept parameters and return values to other procedures. Parameters are passed by value so the only way to return values from one procedure to another is to use global variables or message channels.

Each procedure in PROMELA can be viewed as a process. Therefore, as far as we are talking about PROMELA, we will use the terms procedure and process interchangeably. PROMELA uses the following syntax to define the name and the type of a procedure:

proctype <process-name> (<parameter-list>) **{**

statement;

...

statement

}

Statements are separated by the use of a semicolon *';'* or an arrow *'->'*. The arrow is preferred to the semicolon in those occasions where the causal relation between two statements should be informally declared.

3.4.1 The initial process

In order to instantiate a process, the user has to explicitly execute it through the use of the *run* operator. The statement

run process-name()

will execute a process with the name *process-name*. The *run* statement is always executable, returns the pid of the process and it does not wait for the process to terminate. The first process to be executed is a special process named *init* that plays the same role as process *main()* in C.

3.4.2 Atomic sequences

In PROMELA the user is also equipped with another useful command. The user may define that a sequence of statements should be executed as one, without any interleaving, by the use of the keyword *atomic*. Any number of statements included inside the atomic structure will be executed without any interruption from other processes. Attention must be paid to the fact that the user has to make sure that all the statements inside an atomic sequence are executable, and do not wait for a returned value from some other process. If this is the case, then the sequence will be blocked and the program will terminate abnormally.

3.5 Message channels

Message channels are data structures that provide a method for transferring data from one process to another.

By declaring that a variable is of type *chan*, e.g

chan a=[5] of {byte, int}

the user can use the variable *'a'* to store up to 5 messages that consist of two fields: a byte and an integer field.

Any process that is inside the scope of variable *'a'* can store a message, e.g

a!'c',512

stores a message consisting of the byte field *'c'* and the integer field *'512'*, or, on the other hand, retrieve a message, e.g

a?var1,var2

Channels can be passed as parameters in procedure calls. They provide the means for communication between the callee and the caller. Whenever a message is placed in the channel, it is made available to all the processes that have access to that channel.

Channels pass messages in a first-in first-out order. The send operation is executable only when the channel addressed is not full, while the receive operation is only executable when the channel is non-empty. The user can set another condition on the executability of the receive operation by defining some of the fields as constants, in this way forcing the receive statement to execute only if these constants match the value of the corresponding field in the message that is at the head of the queue.

In order to avoid undesirable blocking, the user can test for the existence of messages before trying to collect them. For example, the statement:

a?[var1,var2]

is a boolean expression that returns true or false depending on whether or not the channel has that message at its head. However, the user must make certain that, in case the message exists, he will read the message before someone else catches it first. This can be ensured via the use of an atomic sequence.

Users can also use channels to achieve synchronous communication. This is achieved by declaring a message channel that can hold 0 messages, e.g.

chan rendezvous=[0] of {*byte*}

This way the user actually defines a rendezvous port that can only pass and not store messages. Unfortunately, only two partners can be synchronised by this method. It will be shown later that this is a major drawback when it comes to the translation of LOTOS specifications.

3.6 Control Flow

PROMELA also provides the user with three different ways for defining control flow:

1. *Case Selection*

   For example, the user may write the following piece of code in order to demonstrate the possibility of a choice between two alternatives:

   ```
   if
   :: (a==true)-> <alternative1>
   :: (a==false)-> <alternative2>
   fi
   ```

   The alternative statements are executed only if the first statements (which are called *guards*) are executable. If both of the *guards* are executable at the same time, the alternative statement to be executed is either of the two. The choice between the two is a non-deterministic one.

2. *Repetition*

   A similar structure can be used to model repetition. Instead of the **if/fi** pair the **do/od** is available in order to provide means for a loop. The program exits from the loop when a 'break' statement becomes executable. In the following example the program exits from the loop when the *guard* statement of *break* becomes executable, i.e when *a==true*.

   ```
   do
   :: (a==false)-> printf("Hello")
   :: (a==true)->  break
   od
   ```

3. *Jumps*

   They provide an alternative to the *break* statement. Instead of exiting a

loop through a *break* statement the user may define a jump to a predefined piece of code by the use of the **goto** command. Labels are used in order to define such pieces. In the following piece of code, the control will *jump* to the label *State_1:* when boolean variable *a* becomes true:

```
                do
                :: (a==false) -> a=true
                :: (a==true) -> goto State_1
                od;
State_1:
                ...
```

3.7 Timeouts

Timeouts are used to escape from hang states. Timeouts are extensively used in computer systems, e.g computer networks, and they prevent systems from hanging.

The *timeout* construct of PROMELA although does not use the notion of real time, characterises states where the control should be transferred in case no other transition is possible. The programmer can make sure that whenever a time-out statement becomes active, reset statements follow, together with appropriate *warning* messages to the user.

3.8 Correctness Criteria

In order to validate a design it is important to define what exactly is a "correct" design. A design can be proven to be correct only with respect to specific correctness criteria.

PROMELA provides us with a mechanism for defining that certain behaviours are impossible. Any protocol which violates any of the claims made by the above

mechanism is considered to be "incorrect". The following structures have been provided by PROMELA for defining impossible behaviours.

3.8.1 Assertions

The PROMELA statement:

assert(cond)

is always executable and can be placed anywhere in a program. The above statement claims that the boolean condition *'cond'* should never be true whenever the *assert* statement becomes executable. The user may also place *assert* statements in a special *monitor()* process, declaring that in any state of the program, *'cond'* should never hold.

3.8.2 Deadlocks

The user can also specify end states by placing an *end*-state label before any statement which may lead to the termination of the program. If the program terminates at any other statement then the termination is considered as an improper one, and the program is said to be *deadlocked*.

End-state labels are strings with the prefix *end*.

3.8.3 Non-progress cycles

In PROMELA we can also mark states that must be executed for the protocol to make progress by using the label *progress* or any other string that starts with that word. If the protocol makes infinite cycles through states that do not make progress, then the protocol is considered to be incorrect.

3.8.4 Livelocks

PROMELA has the ability to claim that something cannot happen infinitely often. By preceding an acceptance-state label before some statement we claim that this

statement may not be part of a sequence of statements that can be repeated an infinite number of times. A program that cycles through the same series of states infinitely, is said to be *livelocked*.

Any label starting with the string *accept* is an acceptance-state label.

3.8.5 Temporal claims

The last and most powerful mechanism that PROMELA provides for the check of the validation of a system is that of temporal claims. The PROMELA notation for a temporal claim is:

never { ... }

A temporal claim defines an order of statements that is impossible to occur. By the term *order of statements* we mean the sequential order where each statement is executed *immediately* after the other (and not just *eventually after*).

The claim is usually a finite state machine. Whenever an undesirable behaviour has occured, the automaton reaches its end (acceptance) state where it cycles infinitely. For example:

```
never {
       do
       :: !flag-> skip
       :: flag->
            accept_label:
                  skip
       od;
 }
```

is an automaton that reaches its acceptance state whenever variable *flag* becomes true. The claim says that it is an error for that variable to become true.

Temporal claims combined with assertions, progress-state labels or acceptance state labels enable us to catch more types of errors. However, temporal claims

increase enormously the complexity of validation programs and therefore they are rarely used.

3.9 A simple example

In chapter 1 (section 1.4) it was stated that: *"In order for the verification procedure to understand and analyse a specification, both the protocol and its properties have to be written in some predefined notation"*.

In this application, this predefined notation is PROMELA.

PROMELA is the language that SPIN (the validation tool used in this project) understands. It can be used to express both the specification and the properties of a protocol. An example of this is presented below:

Suppose we want to specify a buffer that can hold up to k elements. One way to do this is to specify k interconnected buffers that can hold up to 1 element each [Mil89]. We can specify that in PROMELA as follows ($k = 2$ in this paradigm):

The specification for the one element buffer will look like:

```
proctype buffer(chan in,out)
     {
          int value;
          do
          ::in?value;
                out!value
          od
     }
```

The buffer just copies elements from its input to its output.

But we want to create a buffer that can hold more than one element. To do that all we have to do is to connect the output of one buffer to the input of the other. For a two element buffer we should then have:

```
init
   {
          chan in=[0] of {int};
          chan out=[0] of {int};
          chan hide=[0] of {int};
          atomic {
              run buffer(in,hide);
              run buffer(hide,out)
          }
   }
```

This concludes the specification of a two place buffer.

Finally, following LOTOS ideas, an extra process that plays the role of the environment feeding the buffer with elements and taking the elements the buffer outputs could be added. In this case the elements could be numbers. If we feed the buffer with a number, then we increase by one the number we 'll put in next time. If we take something out of the buffer then we decrease the number we input by one. So the specification of the environment will look like:

```
proctype environment(chan in,out)
{
        int value;
        value=1;
        do
        ::in!value;
                value=value+1
        ::out?value;value=value-1
        od
}
```

Furthermore, we should add a line that initiates the environment process before anything else. So, the *atomic* statement in the **init** process will look like:

```
        atomic {
                run environment(in,out);
                run buffer(in,hide);
                run buffer(hide,out)
        }
```

But, is the specification correct? To prove that the specification is correct all we have to do is to prove that the environment never inputs a number that is greater than 2, because that would mean that it would input more than two numbers without taking out any. In other words, that would mean that the buffer can hold more than two elements. In PROMELA the proof is easy. All that has to be done is the addition of an *assert* statement that checks that the numbers the environment inputs are indeed not greater than 2, *in any case*. So, the specification of the environment together with the specification of the properties we want to check is:

```
proctype environment(chan in,out)
{
        int value;
        value=0;
        do
        ::in!value;
                assert(value<3);
                value=value+1
        ::out?value;value=value-1
        od
}
```

Thus, it has been shown how PROMELA can be used to specify both a protocol and its expected behaviour. Other constructs could be used to prove that the buffer eventually inputs a number greater than 0.

3.10 Epilogue

Although we did not mention all the facilities of PROMELA e.g macro definitions, we made clear that it is a powerful validation language.

Designers can fairly easily write any protocol they want to verify and properties the protocol must have. Of course, one must bear in mind when writing that not all the protocols and their properties can be effectively verified, no matter what the language constructs are.

Chapter 4

An overview of SPIN

4.1 Prologue

Protocol verification plays a crucial role in protocol design. Even the most carefully developed protocol specifications may contain errors that will only be detected after exhaustive analysis. The role of a validator is to carry out this analysis and intelligently detect and report any pitfalls the specification may have.

During the years, a lot of automated methods for verification have been developed. SPIN is one such validator that encapsulates most of the available algorithms for protocol verification. It accepts as input a specification written in PROMELA, performs some analysis on this specification and reports any errors. The algorithms and the way SPIN works will be explained below.

4.2 Validation algorithms

SPIN's validation algorithms are based on reachability analysis. SPIN tries to analyse all the possible execution sequences starting from a given initial state.

Reachability analysis algorithms can be categorised into three main types:

- *Full state search*

This method works well for specifications which are not likely to produce more than 100,000 states.

- *(Controlled) Partial search*

  The method is intended for larger systems, that can not be handled by the full state search algorithm. However, this technique has an upper limit and does not work for systems with more than 100,000,000 states.

- *Random execution*

  This technique is used when both of the previous methods have failed. It is intended for huge systems whose state space is too large to be explored.

The performance of a reachability analysis algorithm is measured according to its *coverage* and its *quality*. These quantities are defined as follows:

$$coverage = \frac{states\ searched}{actual\ number\ of\ states}$$

$$quality = \frac{errors\ found}{actual\ number\ of\ errors}$$

4.2.1 Full State Search Method

The algorithm tries to detect which states of the system are reachable and which are not. Unreachable states are pieces of un-executable code in the specification, or error states. Then it tries to analyse all the possible execution sequences.

Any executable sequence of reachable states must be free of deadlocks, of livelocks or of any other criteria the user has imposed.

The obvious problem of the full search method is that it performs well only if the number of states produced by the analysis of the specification is smaller than the number of states the system can store. Moreover, when the number of states produced exceeds the memory capacity of the system the *quality* of the full search method tends to decrease dramatically, together with the *coverage*.

Unfortunately, most of the working protocols today, generate a state space that is too large for exhaustive analysis. Therefore, the full state search algorithm can be used only for relatively small specifications.

4.2.2 Controlled partial search

The controlled partial search method is based on the fact that in most cases it is impossible to fully search the state space produced by a specification. The central idea is that we can pay off this drawback of low *coverage* by maximising the *quality* of the search.

Controlled partial search analyses a number K of states while it makes sure that this number is much smaller than the actual number of states produced.

This can be achieved by many different techniques. There are algorithms that put an upper limit to the length of the execution sequences they analyse (*Depth Bound*). Others try to 'guess' which execution sequences may lead to deadlocks and they analyse only these sequences (*Scatter Search*). Some algorithms analyse the execution sequences that are more likely to happen (*Probabilistic Search*) while others try to drop sequences that are not relevant in the search for error states (*Partial Orders*). Finally, efforts are made to develop algorithms that proceed to the validation of an execution sequence according to some cost function that calculate based upon the states of the sequence (*Guided Search*).

SPIN uses another technique based on *random selections* of execution sequences. The algorithm promises to be more effective than the ones mentioned above and it guarantees that the *quality* of the search will be the maximum possible. This algorithm, called the *super trace algorithm* tries to organise a memory by using all the available bytes and performing the largest search possible. To determine whether a state s is a member of set A the method uses a hash table where it stores information on whether a certain state s has been processed or not. It minimises the amount of memory used, by using a hash table with a very large number of buckets (10^8). Thus, it is possible in most cases to determine a state

if you know the bucket it lies in. This fact decreases the amount of information needed for describing a state linked with a certain bucket and therefore reduces the amount of memory the validator uses. For a discussion on this method the reader can refer to reference [JH91].

4.2.3 Random simulation

The method is used for huge systems, where the state space may be over 10^8 states. The technique will work even for an *infinite* number of states. The state space is explored by a random walk over the transitions of the system. The user can try the algorithm for more than one time, collecting this way more data for his specification.

Although the coverage of the method is low, it has been proven by Colin West that the quality can be, in some cases, adequate [JH91].

4.3 Other algorithms

The algorithms explained so far are used by SPIN to decide which execution sequences should be checked and which assertions have been violated. For the detection of other errors in these sequences SPIN uses a number of different techniques.

To detect the presence of *non progress* and *acceptance cycles* and *temporal claims* SPIN uses the following algorithms.

Detection of non progress cycles

The algorithm for the detection of non progress cycles used by SPIN performs a depth first reachability analysis. When a progress state is reached the search stops. If during the process a cycle is detected then this means that it is a non-progress cycle.

Of course there is a possibility that a non-progress cycle appears after a progress cycle has been seen. To take care of this case SPIN creates another

state space that consists of the original one but where transitions from progress states are disabled. The algorithm checks if it can create a cycle by combining every possible prefix of a cyclic sequence of the initial space with the ones in the second state space.

Detection of acceptance cycles

The algorithm used by SPIN is actually the one presented in [CVY90]. According to [CVY90, JH91] the algorithm uses the depth-first algorithm to traverse two state spaces (that they are initially the same). Let the two state spaces be named A and C respectively. Then for every acceptance state removed from the work space W and added to set A the algorithm swaps A and C and starts searching again. If during this search, the same acceptance state (called the *seed state*) is revisited then an acceptance cycle has been found. If the seed state is not visited again then the second search terminates when all the successors of the seed state have been added to C. Then the algorithm starts again using set A this time.

For a detailed description of the algorithm as well as a proof that this algorithm finds indeed all the acceptance cycles the reader should refer to [CVY90].

Temporal Claims

Temporal claims are usually easy to check. A temporal claim, as mentioned in the previous chapter, is actually a finite state machine. So, to check a termoral claim, all we have to do is to try to match every transition of the automaton that represents the claim with a transition in the state space.

The algorithm goes as follows:

For every new state the search enters, we try to make a transition in the finite state machine that represents the temporal claim. If the transition can not be made then the search is truncated as in the case where a state match occurs. If no transition from the initial state space can be matched

by a a transition in the finite state machine of the claim, then the behaviour the claim represents is impossible.

4.4 Using the validator

The validator can be used to make either a random simulation, or an exhaustive search or a controlled partial search.

For a random search, a statement of the form:

spin <specification file>

where a *specification file* is a specification written in PROMELA, will produce after a while a compact report on the number of process created, the states (if any) where the process has been blocked and other useful information. Different parameters given in the command line will generate more information.

If we want to produce an analyser for an exhaustive or a partial search the validator has to be called as follows:

spin -a <specification file>

This will produce a file named *pan.c* that contains the analyser for the specification. This is a program written in C and must be compiled in order to produce executable code.

The compilation flags determine the type of the search (exhaustive or partial) the analyser will make. A compilation without any flags, e.g:

cc -o pan pan.c

produces an exhaustive search analyser, while on the other hand a *-DBITSTATE* option produces an analyser that performs a controlled partial search. The user can also define upper limits for the memory used, the maximum search depths and other useful parameters.

As soon as the analyser has completed execution and independently of the search method followed, it will report whether it has found any errors, the number of states searched and the total number of states together with some other information.

In the event of an error, a file called *pan.trail* is produced with the sequence of the steps that leaded to it. The user can follow this sequence by instructing *spin* to follow the trail written in *pan.trail*, using the command line option of *spin*: **-t**.

4.5 Validation Techniques

Although controlled partial search can handle systems with up to 10^8 states, most of the time this is not enough. Protocols of realistic complexity produce a number of states that is some magnitudes larger than 10^8. Therefore, it is necessary that protocols are written in such a way that the validation will be the easiest possible.

One method is to define a protocol in different levels and try to validate each level at a time. Protocols may have parts that are completely independent from one another, so a validation made in different stages with each stage handling just one part of the specification is possible.

The same technique can be used if we specify a protocol step by step verifying each time a different property of it.

For example a verifier for all the three lowest level of the OSI architecture would have to explore some billions of states. But the case is simpler if we try to validate a level at a time, and then, assuming that the protocols describing the internal functions of the three levels work properly, try to verify that the protocol for the interactions among them works with no errors.

In addition, the specifier must try to minimise the number of different execution sequences of the system. Usage of *atomic sequences* tends to decrease the interleaving factor, leading to a smaller number of different states the analyser has to explore.

4.6 Epilogue

Although the progress made in the area of verification the last years is great, the problem of verifying large protocols is still open.

It is proven that the problem of analysing the state space produced by a specification is a *PSPACE* problem. Efforts have been made to obtain algorithms that can be applied to a large class of problems, but so far the range of realistic protocols that can be verified remains rather small.

Chapter 5

Translation of basic LOTOS

5.1 Prologue

In this chapter, the process of generating extended finite state machines (XFSMs) from basic LOTOS specifications will be described. The protocol which specifies the way XFSMs are synchronised with each other will be explained in subsequent chapters.

The work is based mainly on two papers. In cite [Kar92] a method is described for deriving extended finite state machines for full LOTOS specifications. A similar work is presented in reference [VSC90], but the execution model is somewhat different. An intermediate model for LOTOS specifications is also presented in cite [AMnS92].

The execution model we have followed is closer to that of reference [VSC90] although some ideas have been taken from reference [Kar92] as well.

The first section of the chapter briefly describes the lexical and syntax analysis phase of the translator. After the syntax analysis phase, abstract syntax trees are created. This process is described in the third section of the chapter. Section four provides details on the method irregular process are identified. The procedure for generating extended finite state machines from abstract syntax trees is described in the fifth section. Finally, the last part has information on the method followed

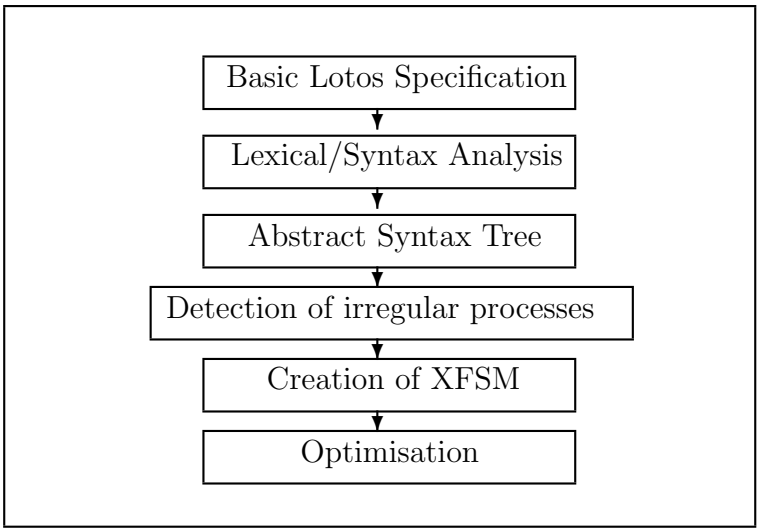

Figure 4: Path to implement the translator

for the optimisation of the finite state machines produced.

In Figure 4 the path followed to produce finite automata from LOTOS specifications is shown.

5.2 Lexical and Syntax Analysis

The tools used for the lexical and syntax analysis phase of the translator are the widely known programs *Lex* [ELS75] and *Yacc* [CJ78].

- *Lexical Analysis*

  The translator accepts as input any basic LOTOS expressions, as presented in cite [BB87]. Anything inside (* ... *) is considered to be a comment and therefore is disregarded.

- *Syntax Analysis*

  The syntax analysis will efficiently parse basic LOTOS specifications with only one exception. Procedures must not be nested. The unfolding of

nested procedures can be done through a special 'pre-compiler' process that is described in reference [ISO89a].

The process is called *flattening* of a specification and it is intended for full LOTOS. However, it can be applied to basic LOTOS as well. Among the main functions of this procedure is the renaming of identifiers (so that each one will become unique), the separation of the data from the control part and the unfolding of nested procedures.

So the translator's input may be specifications that look like:

```
specification spec_name [gate list] :functionality
    behaviour
          behaviour expression
    where
          process definition
          ...
          process definition
endspec
```

A *process definition* looks like:

```
process proc_name [gate list] :functionality:=
   behaviour expression
endproc
```

The context free grammar used in the syntax analysis phase is based on the one presented in [ISO89a]. The difference is that, since the parser is intended for basic LOTOS, any production containing data types has been removed. The grammar can be found in Appendix A.

- *Symbol Table*

The symbol table that holds information about the identifiers used in a specification (such as gate and process identifiers) is a hash table that consists of other hash tables [PPP91].

Each process has its own table. Identifiers declared in the specification body can be found in the root table. In addition, each time a **hide** operator is met, a new child hash table is generated. This way scope information is easily represented. The identifiers a behaviour expression can access are the ones declared in the current and the 'father' hash tables. Although scope rules in basic LOTOS are not so complex, the symbol table was designed to handle full LOTOS specifications. In Figure 5, an example on the way information is represented in the symbol table is shown. The example illustrates the state of the symbol table after a specification of the form:

```
specification spec_name[a,b]:exit
    behaviour
        ....
    where
        process proc1_name[c]:exit:=
           ...
          hide k,l in
             ...
        endproc
        process proc2_name[a,c]:noexit:=
             ...
        endproc
endspec
```

has been analysed.

Furthermore, to make sure that gates and processes have unique names, a unique number is assigned to each gate and process every time a new

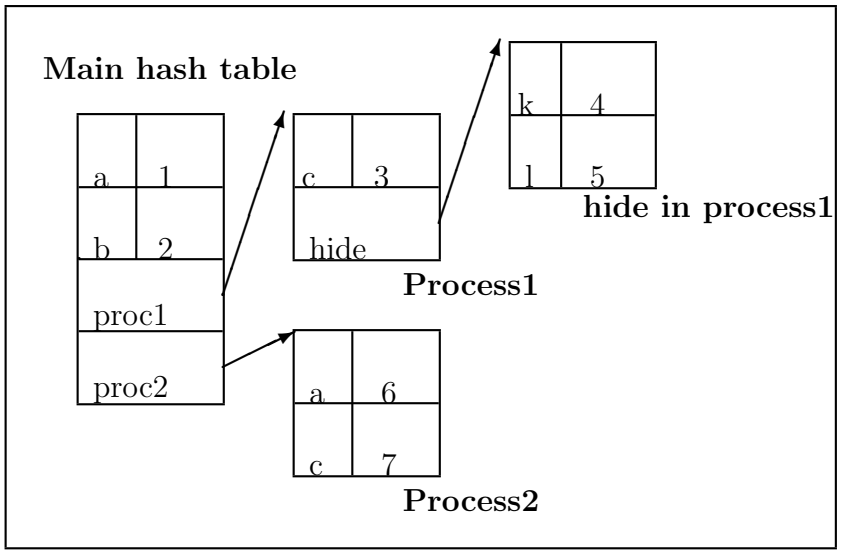

Figure 5: Structure of the symbol table

identifier is declared. Therefore, after the syntax analysis, each gate consists of a name (its lexeme) and a number. It is likely that the code generated from the LOTOS specification will sometimes refer to a gate by its number and not by its name.

5.3 Creation of Abstract Syntax Trees

An abstract syntax tree for the specification being analysed is constructed during the syntax analysis phase of the translator.

Abstract syntax trees can be thought as parse trees where each node reflects to an operator [VASDU86]. The branches of a node point to its operands. Abstract syntax trees provide us with an intermediate representation that allow one to further process and transform the input text.

In LOTOS case, we do not maintain just one single abstract syntax tree but a forest of abstract syntax trees. Each tree in the forest represents a process definition in the LOTOS specification.

Trees may point to each other. This is the case when a process instantiation

occurs. A node that represents process instantiation points to the root of the tree that represents the process that is about to be instantiated. This leads us to the creation of a directed acyclic graph (dag).

An example can better demonstrate how abstract syntax trees are created.

Suppose that we have the specification:

```
specification test [] : noexit
     hide in,out in
          buff [in,out]
 where
     process buff [in,out]:noexit:=
       in ;
          buff [out,in]
     endproc
 endspec
```

This specification will produce two abstract syntax trees that will look like in Figure 6.

When a process instantiation occurs before the definition of the process body the above method can not be followed. In this case, the nodes hold just the name of the process to be instantiated while a special field denotes that this node has not been completed yet. The node will point to the correct tree only in subsequent phases when the creation of the forest is completed.

The nodes of an abstract syntax tree usually hold more information than just the operator of an expression. Each node is assigned with a name. This name consists of the name of the node's operator and the names of the children of this node.

Therefore, the node that represents the [] operator of the expression:

$$a; stop \ [] \ b; stop$$

has the name:

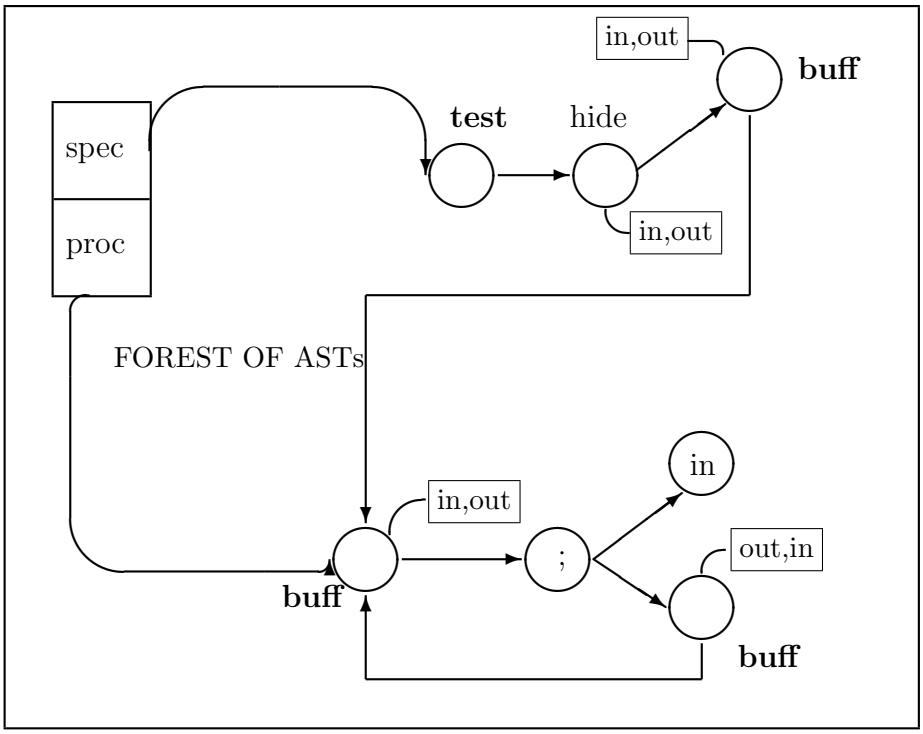

Figure 6: An example of a forest of abstract syntax trees

Node's name: **[];stop;stop**

These names are used as index keys. Nodes are stored in a hash table so they can be easily and quickly retrieved at any time. It should be noted here that, with this scheme, similar expressions are likely to have nodes with the same name. This feature will become useful at the optimisation stage.

Also, at this stage the **exit** operator of basic LOTOS is transformed into the expression:

$$\delta \ ; \ \textbf{stop}$$

Gate δ is considered to be a special gate and it has a unique name. It should not be confused with any other gates with the same name that may be declared in the specification. Letter δ is used to denote the successful termination of a process and has a special role in the sequential composition of processes.

Finally, nodes that reflect operators that take parameters, such as the *parallel* or the **hide** operator have extra fields to hold that information. In addition, at this stage we extend the list of synchronisation gates of a parallel operator so that they will include the *exit* gate: δ. If inside a process with a **noexit** functionality occurs an interleaving operator, (|||), then the interleaved processes are not synchronised at all (since they do not exit).

In Figure 6, this is made clear. Each node that has parameters points to a linked list. If we had a parallel operator then gate δ would be added to that list.

Some semantic errors can be detected during the construction of abstract syntax trees. For example, references to undefined identifiers can be revealed when trying to create nodes for gates that can not be found in the symbol table. At this stage we can also detect processes that have functionalities that do not match. This is the case when we have processes that exit although they have been declared as **noexit** processes.

5.4 Identification of non regular expressions

By the term *non regular expressions* we mean expressions that may lead to dynamic creation of processes which, therefore, can not be modelled by finite state machines. Such expressions are likely to produce an infinite number of states. Thus, these expressions must be identified in order to produce finite automata.

Subsequent phases will handle such expressions properly.

More precisely, if inside a process P_1 that calls another process P_2 exist any of the following schemas:

- $C_1[C_2[P_2] \circ B]$,
- $C_1[B \diamond C_2[P_2]]$, or
- $C_1[par\ g_1, ..., g_n\ in\ [h_1, ..., h_n] \diamond C_2[P_2]]$

where $\diamond$ is the parallel operator and $\circ$ is any parallel operator, enable operator or

the disable operator, B any behaviour expression, C_1, C_2 any pair of syntactical contexts and P_2 calls P_1 then it is proven in [GN89] that the resulting transition system is possibly infinite.

Only the first two expressions are of interest in this translator, since the third one is only found in full LOTOS.

Irregular expressions can be found if the abstract syntax trees are searched for expressions where there is a recursive process instantiation on the left or the right hand of a parallel operator, or a recursive process instantiation on the left hand of an enable or disable operator.

The algorithm for the detection of irregular processes is shown in Figure 7. It is based on one presented by Karjoth [Kar92].

To detect non regular expressions we start searching from the root of the syntax tree that reflects the specification's body. The algorithm will return a list of irregular processes, i.e processes that instantiate themselves on the left (or right) hand side of a parallel, enable or disable operator. Therefore, any expression that contains the instantiation of such a process, is an irregular expression.

While we are searching we maintain two lists:

The first list, *seen_so_far*, keeps track of all process identifiers that occurred before the last parallel composition, enable or disable operator. The second one, *proc_list*, maintains a record of the process identifiers that occurred after the parallel composition.

In an event of a parallel instantiation there are the following possibilities.

The process is encountered for the first time.

> In this case, we have to continue the search in the process' body. We add the process' identifier to the list of the processes met after a parallel, enable or disable operator and we continue searching by jumping to the root node of the syntax tree that represents that process.

The process has been encountered before.

```
                    Detect (node,seen_so_far,proc_list)
switch (node)
    case STOP:
        return;
    case CHOICE:
        return Detect(left_child,seen_so_far,proc_list)
                                ∪
                    Detect(right_child,seen_so_far,proc_list)
    case Action Prefix: case HIDE:
        return Detect(only_child,seen_so_far,proc_list)
    case PARALLEL:
        union_proc_list=Detect(left_child,seen_so_far ∪ proc_list,nil)
                                ∪
                    Detect(right_child,seen_so_far ∪ proc_list,nil)
        if (union_proc_list=nil)
                    mark node
        return union_proc_list
    case ENABLE: case DISABLE:
        union_proc_list=Detect(left_child,seen_so_far ∪ proc_list,nil)
                                ∪
                    Detect(right_child,seen_so_far,proc_list)
        if (union_proc_list=nil)
                    mark node
        return union_proc_list
    case INSTANTIATION:
        if (instantiated process ∈ proc_list)
                    return nil
        if (instantiated process ∈ seen_so_far)
                    return process_name
        else
            return
                Detect(process_root_node,seen_so_far,process_name ∪ proc_list)
```

Figure 7: Detection of non regular expressions

There are two possibilities:

- A "dangerous"[1] operator has not occurred in the meantime.

  Since we had encounter the process before, it means that we are dealing with a recursive process that is about to start executing the same code again. But the code does not contain any "dangerous" operator. Therefore, the expression that contains the instantiation of that process is regular.

- A parallel operator has occurred. This means that the process recursively instantiates itself on the left (or right) of a "dangerous" operator and its name has to be added to the list of irregular processes.

Each time a parallel, enable or disable operator which may give rise to another process is met, the node in the abstract syntax tree is marked. Whenever during the creation of the finite state machines such a node is encountered, a special state will be created commanding the finite state machine to stop execution and generate new finite state machines.

The code needs little explanation for the other cases.

Whenever we meet a parallel operator (||, |||, |[]|), we add the list of processes in *proc_list* to the list of processes that occured before the operator, and we continue the search in the child trees that constitute the operands.

An enable (>>, or disable operator ([>) is handled in a similar way. The difference is that, since only the left part may give rise to dynamic process creation, we do not change the lists *proc_list* and *seen_so_far* for the right part.

The action prefix (**;**) and the **hide** operators can not produce irregular expressions, so whenever we meet such an operator we merely continue with the next node in the tree.

A choice ([]) node similarly makes the algorithm to search its child nodes.

[1] i.e a parallel, enable or disable operator

A **stop** node in the tree marks the end of an expression and causes the algorithm to return. It should be mentioned again that there are no **exit** nodes in a syntax tree, since they have all been translated into δ;**stop** expressions.

Using this algorithm we can mark all the nodes in the forest that dynamically produce new processes. In the next phase every time such a node is encountered, the creation of the finite state machine will stop and a command that generates a new finite state machine (that simulates the behaviour of the irregular expression) will be added.

By *extending* the finite state machines with the ability to generate new automata, we can effectively simulate the behaviour of these expressions.

Let's see an example, where a specification has been syntactically analysed and an abstract syntax tree has been produced. The specification is the following one:

```
specification infinity[start_s,start_a]:noexit
    behaviour
    hide send,receive in
        (start_s;handle_pack[send]
            |||
        start_a;handle_pack[receive]
        )
    where

        process handle_pack[packet]:noexit:=
            packet;
                (send[packet]
                    |[packet]|
                handle_pack[packet])
        endproc
```

```
        process send[packet]:noexit:=
            hide send_pack in (
                packet;send_pack;send[packet]
                    [>
                (hide find_error,report_error in
                    ( abort[find_error]
                            >>
                    report_error;stop))
                )
        endproc

process abort[find_error]:exit:=
           hide not_found,found in
                  find_error;
                  ( not_found;abort[find_error]
                      []
                    found;exit
                  )
        endproc
endspec
```

The specification contains three expressions that are irregular. These are the two expressions that contain the parallel operator, and the expression containing the disable operator. Note that the left hand of the enable operator is not an irregular expression although it recurses at some point.

The translator will produce the following syntax tree:

```
SPECIFICATION infinity [ start_s,start_a]
         BEHAVIOUR
```

```
               HIDE send,receive IN
       start_s ;handle_pack[send]
       |[ NONE ]|
       start_a ;handle_pack[receive]
(end tree)

PROCESS handle_pack [ packet ]
        BEHAVIOUR
               packet ;send[packet]
       |[ packet ]|
       handle_pack[packet]
(end tree)

PROCESS send [ packet ]
        BEHAVIOUR
               HIDE send_pack IN
       packet ;send_pack ;send[packet]
           [>
       HIDE find_error,report_error IN
            abort[find_error] >> report_error ;STOP
(end tree)

PROCESS abort [ delta,find_error ]
        BEHAVIOUR
               HIDE not_found,found IN
            find_error ;not_found ;abort[find_error]
                []
            found ;delta ;STOP
(end tree)
```

Words in uppercase letters are operators representing interior nodes in the tree. Words in lowercase (like gate names) represent leaves. Notice that the **exit** operator has been transformed into a

delta; STOP

where *delta* represents the letter δ. It is also shown how δ has been added to the parameter list of process *abort*, which is a process with functionality: **exit**.

Also notice that the two *handle_pack* processes that run interleaved, have no synchronisation points. This is because both processes have a **noexit** functionality. Had the functionality been **exit** the processes will be synchronised at gate *delta*, i.e δ.

After the algorithm for the detection of irregular expressions has been applied we get the following:

```
SPECIFICATION infinity [ start_s,start_a,]
         BEHAVIOUR
                HIDE send,receive IN
        start_s ;handle_pack[send]
        |[ NONE ]| (IRREGULAR)
        start_a ;handle_pack[receive]
 (end tree)

 PROCESS handle_pack [ packet ]
         BEHAVIOUR
                packet ;send[packet]
        |[ packet ]| (IRREGULAR)
        handle_pack[packet]
 (end tree)

 PROCESS send [ packet ]
```

```
         BEHAVIOUR
                HIDE send_pack IN
        packet ;send_pack ;send[packet]
        [> (IRREGULAR)
       HIDE find_error,report_error IN
        abort[find_error] >> report_error ;STOP
(end tree)

PROCESS abort [ delta,find_error ]
         BEHAVIOUR
                HIDE not_found,found IN
          find_error ;not_found ;abort[find_error]
                    []
          found ;delta ;STOP
(end tree)
```

Clearly, the algorithm has correctly identified any irregular expressions. In the next section, the method of creating extending finite state machines out of this annotated syntax tree will be described.

5.5 XFSM creation

Finite state machines created by the translator are extended in the following sense:

- A finite machine can call a new finite state machine.
- At each state, a set of variables may be defined or updated.

For the moment, we will consider that these finite state machines have the means to communicate with each other, although the communication medium is not shown anywhere.

5.5.1 State Machines

A finite state machine can be viewed as an automaton that consists of states and transitions. The automaton starts from a predefined state (*initial*) state and moves according to some predefined rules (*transition rules*). In this model a transition rule can be represented as:

$$s_n \xrightarrow{\alpha} < s_{n+1}, c >$$

This symbolism can be translated like that:

The state machine was in state s_n, offered an event at gate α and entered state s_{n+1}. There, it updated its variables by executing command c.

Each machine has a unique name, which belongs to the set of machine names $\mathcal{N}$ and an initial state s_0.

Commands may be of the following kinds:

1. Declarations

   Each state machine can access a predefined set Σ of variables, that we will call *gates*. They constitute the alphabet of the machine. Declarations are used to extend the set of gates a finite state machine can access.

2. Assignments

   By an assignment statement we state the equivalence of two gates. This means that a statement of the form:

   $$a = b$$

   denotes that gates a and b refer to the same thing. a copies any property b has. In the next chapter, it will be shown how the synchronisation algorithm associates each gate with a set of properties. Moreover, the above assignment contains an *implicit* declaration of gate a.

3. Finite State Machines synchronisation call

A call to a finite state machine can be viewed as a procedure call. A finite state machine call causes the automaton which executed the command to stop and the machine that was called to start executing. A synchronisation call consists of two process names and a set of gates where the processes must synchronise.

4. Finite State Machines disabling call

   It is similar to the previous one, but this time processes are not synchronised. However one process may cause (non deterministically) the termination of the other one. Control is transferred to the process that commanded the interrupt.

5. Finite State Machines enabling call

   It consists of two calls. The first call instantiates an automaton that as soon as it reaches a proper end state causes the automaton instantiated by the second call to start execution.

So the set $\mathcal{C}$ of commands has elements of the following form:

- **new** $v_1, ..., v_n,\ v_1, ..., v_n \in \Sigma$
- $a = b\ a, b \in \Sigma$
- **Pcall** $name1(v_1, ..., v_n)$,$name2(n_1, ..., n_k),\ g_1, ..., g_m$

  $name1, name2 \in \mathcal{N},\ v_i, n_j, g_l \in \Sigma, i = 1..n, j = 1..k, l = 1..m$
- **Dcall** $name1(v_1, ..., v_n)$,$name2(n_1, ..., n_k)$

  $name1, name2 \in \mathcal{N},\ v_i, n_j \in \Sigma, i = 1..n, j = 1..k$
- **Ecall** $name1(v_1, ..., v_n)$,$name2(n_1, ..., n_k)$,

  $name1, name2 \in \mathcal{N},\ v_i, n_j \in \Sigma, i = 1..n, j = 1..k$

The above discussion leads to the following definition of the extended finite state machines the translator produces.

Definition 1 *An extended finite state machine, or XFSM is a 6-tuple* $M =< S, \Sigma, N, C, R, s_0)$, *where:*

- S *is a finite set of control states.*
- Σ *is a finite set of gates.*
- N *is a finite set of process names.*
- C *is a finite set of commands.*
- R *is a finite set of rules.*
- s_0 *is the initial state of* M.

R, *the set of rules is a finite subset of* $S \times \Sigma \times C^k \times S$, *where* $k \in [0, ..., |C|]$.

The set of machines $\mathcal{M}$ that simulates the behaviour of the LOTOS specification constitutes a network.

When two state machines start execution due to a synchronisation call then they synchronise at the transitions that involve gates belonging to the synchronisation set. Moreover, any state machine created at subsequent steps by any of the initial ones, is considered to synchronise at least at the same gates as its father.

In a disable call we consider that one of the machines (the one that appears to be second in the call) can instruct at any time the other automaton (or its children) to stop, if that automaton (or any of its children) is still executing. In this case the machine that instructed the stop starts execution. Before the disable instruction the machine is considered to be *inactive*.

We can think of each instantiation (*call*) of each finite state machine as a representation of a LOTOS process. The behaviour of the specification translated is simulated by the joint behaviour of the instances of the finite state machines [VSC90].

Normally, since each machine can be *called*, an initialisation command c_0 would be needed. However, each machine will be translated into a PROMELA process, so this statement will be executed implicitly by parameter passing and there is no need to explicitly define it. For this purpose, we have associated each finite state machine with a set of parameters that decides the initial set of gates Σ to which the automaton has access. The value assigned each time to these parameters changes according to the values passed when the machine is *called*.

This model of extended finite state machines closely reassembles similar models presented by Karjoth, or Valenzano *et al* [Kar92, VSC90].

Hereafter, the process of translating each operator of basic LOTOS into a finite automaton will be explained. The translation procedure is based on the one presented in cite [Kar92]. Finite state machines for LOTOS processes will be created by composing the automata produced for each operator.

5.5.2 Inaction: stop

The **stop** operator represents the inactive process. Therefore, it can be simulated by an automaton with just one state which is not able to make any transitions. The automaton can be defined as:

$$M(\mathbf{stop}) = < \{s_0\}, \emptyset, \emptyset, \emptyset, \emptyset, s_0 >$$

Intuitively, **stop** represents *end* states in the automata that simulate LOTOS processes. Once a process reaches an *end* state, it can not continue further.

However, **stop** denotes an improper termination of a process. Proper final states are denoted in LOTOS terminology by the **exit** operator. Since **exit**s have been transformed into δ;**stop** expressions, we consider the δ;**stop** construct to be a special case of a stop expression, that denotes successful termination. δ;**stop** is translated like the **stop** operator and not like the action prefix which is described below. Thus,

$$M(\delta;\mathbf{stop}) = < \{s_0\}, \delta, \emptyset, \emptyset, \emptyset, s_0 >$$

5.5.3 Action prefix

In LOTOS the expression:

$$g; B$$

means that if a process can synchronise (with other processes or with the environment) at gate g, then it will transform into the behaviour expression B. We can simulate that by a finite automaton with two states. The automaton can offer an event at gate g and move from the first state into the second. The second state is the initial state of the automaton of expression B. So:

$$M(g;B) =< \{s_0, s_1\}, \{g\}, \emptyset, \emptyset, \{s_0, g, s_1, \emptyset\}, s_0 >$$

where s_1 is the initial state of $M(B)$.

The *internal* action **i** is handled in the same way.

Action prefix and **stop** are the basic components out of which more complex machines that simulate the rest of the operators of basic LOTOS are built.

5.5.4 Choice

A choice expression is of the form:

$$B_1 \; [] \; B_2$$

and it states that a process may act either like B_1 or B_2. The first action is the one that determines which expression will be followed. An automaton for a choice expression is composed by the automata for the expressions B_1, B_2. The initial state of $M([])$ is a new state not belonging to the state sets of $M(B_1), M(B_2)$. The transitions of the initial states of $M(B_1), M(B_2)$ become transitions of the initial state of $M([])$. In Figure 8, the method for constructing an automaton for the choice expression

$$a; b; stop \; [] \; c; d; stop$$

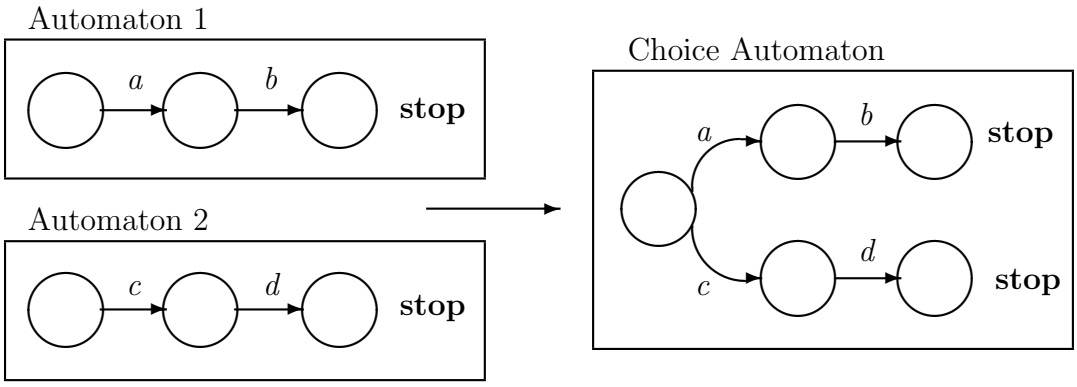

Figure 8: An automaton for a choice expression

out of two existing ones, is shown.

If,

$$M(B_1) =< S_1, \Sigma_1, N_1, C_1, R_1, s_{0_1} > \text{and}$$
$$M(B_2) =< S_2, \Sigma_2, N_2, C_2, R_2, s_{0_2} > \text{and}$$
$$S_1 \cap S_2 = \emptyset$$

then we can define the finite state machine for the choice expression as being the following one:

$$M([]) =< S, \Sigma, N, C, R, s_0 >$$

where: $s_0 \notin S_1 \cup S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $N = N_1 \cup N_2$, $S = (S_1 - \{s_{0_1}\}) \cup (S_2 - \{s_{0_2}\} \cup \{s_0\})$.

If $R(s)$ is the set of rules at a given state s then the set of rules for $M([])$ can be defined as follows:

$$\forall s \in S, s \neq s_0, R(s) = R_1(s), \text{ if } s \in S_1 \wedge s \neq s_{0_1}$$
$$\forall s \in S, s \neq s_0, R(s) = R_2(s), \text{ if } s \in S_2 \wedge s \neq s_{0_2}$$
$$R(s_0) = R_1(s_{0_1}) \cup R_2(s_{0_2})$$

5.5.5 Hiding

The hiding operator is not translated into a finite state machine. The expression:

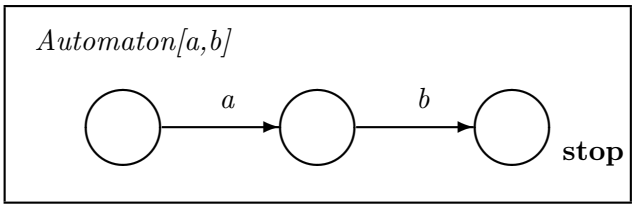

Figure 9: An automaton without declarations

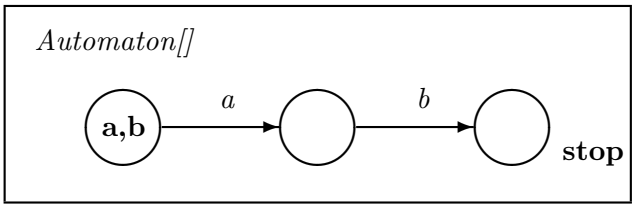

Figure 10: An automaton with declarations

hide $g_1, ..., g_n$ **in** B

will cause a *declaration* command to be added at the initial state of the automaton that represents B. This means that a specification

```
specification NoHide[a,b]:noexit
behaviour
      a;b;stop
endspec
```

will produce the automaton in Figure 9, while a similar automaton (but with a different set of commands) will be produced by the next specification (Figure 10).

```
specification Hide[]:noexit
behaviour
    hide a,b in
      a;b;stop
endspec
```

If

$$M(B) =< S, \Sigma_1, N, C_1, R_1, s'_0 >$$

is the automaton of expression B, $\Sigma_0 = \{g_1, ..., g_n\}$ and c is the declaration command, then

$$M(\textbf{hide}) =< S, \Sigma, N, C, R, s_0 >$$

where $\Sigma = \Sigma_0 \cup \Sigma_1$, $C = C_1 \cup \{c\}$.

If $R_1(s)$ is the set of transitions from a state s then we can define set R as being:

- $\forall s \in S, s \neq s_0, R(s) = R_1(s)$
- $\forall s_k \in S, \sigma \in \Sigma$, such that $(s_0, \sigma, \{c'\}, s_k)$ is a transition, we define a transition $(s'_0, \sigma, \{c'\} \cup \{c\}, s_k)$

  Then, $R(s_0)$ is the set whose elements are these transitions.

It should be noted that :

$$\forall \Sigma, \Sigma_0 \cap \Sigma = \emptyset$$

That means that each gate in Σ_0 is unique and therefore it can not be used for synchronisation by any other process. Hiding is implemented by this method.

5.5.6 Parallel Composition

LOTOS statement:

$$B_1 \ |[g_1, ..., g_n]| \ B_2$$

states that expressions B_1, B_2 must be synchronised at gates $g_1, ..., g_n$ and at gate δ.

There are two different methods for implementing this.

If expressions B_1, B_2 are regular then a finite state machine can be produced that simulates the synchronised moves of B_1,B_2. Suppose that $M(B_1)$ and $M(B_2)$ are finite state machines simulating expressions B_1 and B_2 respectively.

Let:

$$M(B_1) =< S_1, \Sigma_1, N_1, C_1, s'_0 >$$
$$M(B_2) =< S_2, \Sigma_2, N_2, C_2, s''_0 >$$

be the components of each machine and

$$G = \{g_1, ..., g_n\} \cup \{\delta\}$$

be the set of gates where B_1 and B_2 must synchronise. Also, let $M(B)$ be the automaton that simulates $B_1 \parallel B_2$. Because $M(B)$ must simulate transitions of $M(B_1)$ and $M(B_2)$ in a parallel way, we can think of each state of $M(B)$ as consisting of two elements. The first one represents the state at which $M(B_1)$ is and the other the state of $M(B_2)$. A transition from one state of $M(B)$ to another will actually simulate a move in $M(B_1)$ and a move in $M(B_2)$ made in parallel.

Having that in mind we can easily conclude that the state space of $M(B)$ is the Cartesian product of the state spaces of $M(B_1)$ and $M(B_2)$. Therefore, if S is the state space of $M(B)$ then,

$$S = S_1 \times S_2$$

The set of rules that $M(B)$ must follow should be such that all the possible execution sequences of $M(B_1)$ and $M(B_2)$ are possible. This results to a set of rules that can be defined as follows:

$$R(< s_1, s_2 >) = \{g, \{c\}, < s'_1, s_2 >, \text{if } (g, \{c\}, s'_1) \in R_1(s_1\} \wedge g \notin G\}$$
$$\cup \{g, \{c\}, < s_1, s'_2 >, \text{if } (g, \{c\}, s'_2) \in R_2(s_2\} \wedge g \notin G\}$$
$$\cup \{g, \{c\}, < s'_1, s'_2 >, \text{if } (g, \{c\}, s'_1) \in R_1(s_1),\ (g, \{c\}, s'_2) \in R_2(s_2) \wedge g \in G\}$$

The above equality states that for each transition in $M(B_1)$ and $M(B_2)$ that does not belong in the synchronising set G we add a transition in $M(B)$. For transitions involving gates that belong to G we add a transition in $M(B)$ only if both $M(B_1)$ and $M(B_2)$ can move at the same time.

The other components of $M(B)$ can be defined as follows:

- $N = \emptyset$, otherwise B_1, B_2 would not be regular.

- $C = C_1 \cup C_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$

So, the composed machine $M(B)$ is defined as

$$M(B) =< S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \emptyset, C_1 \cup C_2, R, < s_0', s_0'' >>$$

It is obvious from the previous discussion that a composite automaton that simulates the parallel operator, would have a large state space. Moreover, B_1's and B_2's synchronisation depends upon factors that may change dynamically, if for example the procedure in which B_1, B_2 are, is called with a different set of parameters. Therefore, we have changed the route of implementing a finite state machine for the parallel operator. Nodes containing a $\|$ operator are always marked to be "irregular".

That means, the parallel operator in this translator is implemented in the same way, independently from the "regularity" or not of the expressions B_1, B_2.

This process can be summarised in the following:

Whenever a parallel operator is met we add a new state, let us call it *split state* to the state machine we were constructing and we stop construction.

Whenever the automaton moves to the *split state*, a command is executed running two instances of the states machines that simulate the expressions B_1 and B_2. The automaton then stops execution. Consider the following paradigm:

```
process process1 [a,b] : noexit :=
a;
    (b;process1[a,b]
         |||
    a;process1[a,b])
endproc
```

The translator will produce three finite state machines. Two of them will simulate the expressions:

$$B_1 = b; process1[a, b],\ B_2 = a; process1[a, b]$$

while the other one will behave according to the expression:

$$B = a; (B_1 \ ||| \ B_2)$$

The last one is of interest because it illustrates the resolving of the parallel operator. It is represented below:

```
Start Automaton Process_0 (a,b )

State:1
        Substitute: a,b = a,b,
        a -> goto state 2

State:2
        Process_1 ( b,a)
                PARALLEL in
                        NONE
                with
        Process_2 (b,a)

End_Automaton Process_0
```

Note that the behaviour of the translator would be the same even if B_1, B_2 were regular for the reasons explained above. Also, note that the automaton has associated with it a set of parameters, that are initialised implicitly when the automaton is called. For example, the parameters of automaton *Process_1* will be assigned the values of b and a when the automaton starts execution.

So, the automaton produced for the parallel expression can be defined as:

$$M(\|) =< \{s_0, s_1\}, \Sigma \cup \{\epsilon\}, \{B_1, B_2\}, C, R, s_0 >$$

The components of the machine are defined as follows:

- C=$\{c\}$ where,
  c =**Pcall** $B_1(\alpha_1, ..., \alpha_n), B_2, (\beta_1, ..., \beta_k), g_1..., g_n$
  with $g_1..., g_n$ being the synchronisation gates
- $\Sigma = \{\alpha_1, ..., \alpha_n\} \cup \{\beta_1, ..., \beta_n\} \cup \{g_1..., g_n\}$
- ϵ symbolises the silent move.
- $R = \{(s_0, i, c, s_1)\}$

In the example presented above the silent move is not shown. This is because a transition of the form:

$$s_n \xrightarrow{\alpha} s_{n+1} \xrightarrow{\epsilon} s_{n+2}$$

can be safely reduced into the equivalent form:

$$s_n \xrightarrow{\alpha} s_{n+2}$$

However, there are cases where the addition of a silent transition may lead to improper results. This happens when a state has more than one alternative. Consider an expression:

$$((b; ... \parallel a; ...) \; [] \; a; ...)$$

This will produce a sequence of states like the following one

$$s_n \xrightarrow{\alpha} s_{n+1}, \text{ and}$$
$$s_n \xrightarrow{\epsilon} s_{n+2} \text{ (blocked)}$$

If the automaton follows the path from s_n to s_{n+2} then it will be blocked, although the expression does not contain any deadlocks, since there is always the alternative of choosing the right hand part.

Consequently, we have to restrict the use of silent transitions into states that do not have more than one alternatives. This means that whenever two state

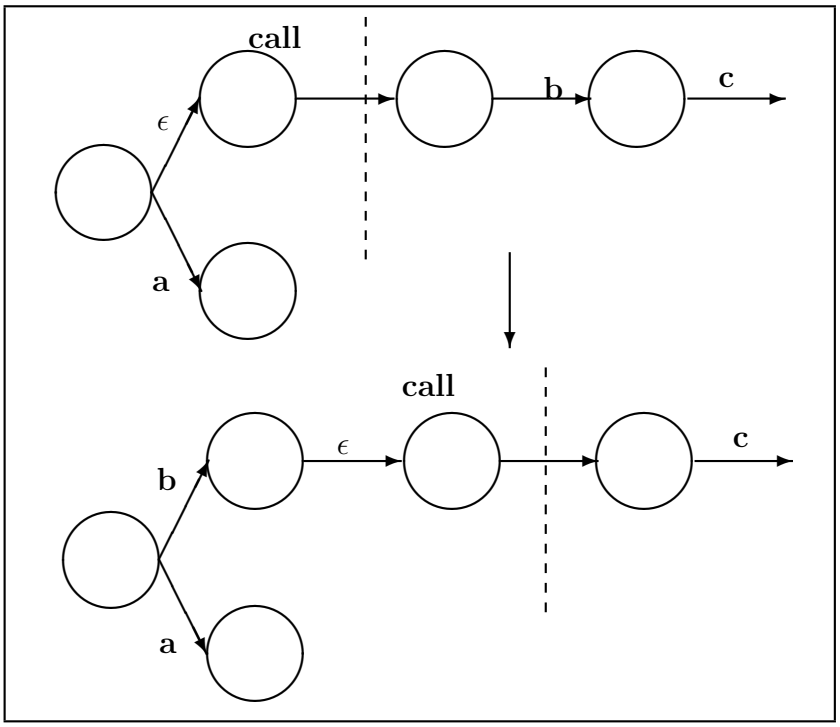

Figure 11: Transforming a choice expression

machines have to be combined in order to simulate a [] operator we have to take care that the initial transitions are not silent.

This is not a severe restriction. Specifications can be easily written in a way first actions of the left and right hand of a [] operator do not lead into a *process call*. On the other hand, we can transform the specification at compile time. This transformation is illustrated in Figure 11.

In this figure, the automaton starts by having two possible moves. The one is silent and leads to a process call. After that call, the automaton which will start execution performs a move by offering b. By pre-calculating the first (actual) move made after the silent transition, we can transform the automaton so that there are no alternatives to the ϵ move.

This first move is the first transition of the automaton produced for the parallel operator. But this method may lead us to recursively trying to calculate which is the first move. To avoid an infinite number of steps, irregular expressions must

be guardedly well defined, in the sense expressed in [Mil80]. This condition is generally used as a guideline when writing LOTOS specifications.

5.5.7 Enabling

The translation of an enabling expression:

$$B_1 \gg B_2$$

depends on whether expression B_1 is irregular or not.

If B_1 is an irregular expression then the translator proceeds in a manner similar to the parallel operator.

Thus, the automaton produced for an (irregular) enabling expression has two states. The automaton moves from the initial to the second one with a silent transition. There it stops execution as soon as instantiates the two automata that simulate expressions B_1 and B_2. So,

$$M(\gg) =< \{s_0, s_1\}, \Sigma \cup \{\epsilon\}, \{B_1, B_2\}, C, R, s_0 >$$

The components of the machine are defined as follows:

- C=$\{c\}$ where,
  c =**Ecall** $B_1(\alpha_1, ..., \alpha_n), B_2, (\beta_1, ..., \beta_k)$
- $\Sigma = \{\alpha_1, ..., \alpha_n\} \cup \{\beta_1, ..., \beta_k\}$
- ϵ symbolises the silent move.
- $R = \{(s_0, i, c, s_1)\}$

Of course, the restrictions applied in the case of the parallel operator are valid here, as well.

If the enabling expression is regular then we can build a finite state machine that behaves according to that expression. To do that we simply have to add a transition out of each (proper) exit state of $M(B_1)$ to the initial state of $M(B_2)$. So if,

$$M(B_1) =< S_1, \Sigma_1, N_1, C_1, R_1, s_{0_1} > \text{and}$$
$$M(B_2) =< S_2, \Sigma_2, N_2, C_2, R_2, s_{0_2} >$$

then the automaton for the enabling expression can be defined as:

$$M(B) =< S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, N_1 \cup N_2, C_1 \cup C_2, R, s_{0_1} >$$

with R being defined as follows:

- $\forall (s_n, g, c, s_k) \in R_1(s)(s_n, g, c, s_k) \in R(s)$,
- For each $(s_n, g, c, s_k) \in R_1(s) \wedge g = \delta, (s_k, \epsilon, c, s_{0_2}) \in R(s)$
- $\forall (s_n, g, c, s_k) \in R_2(s)(s_n, g, c, s_k) \in R(s)$,

Thus, for a specification of the form:

```
process process1 [a,b] : noexit :=
            a; b;exit >> a;b;stop
        endproc
```

the translator will produce the state machine:

```
Start Automaton Process_0 (a,b)

State:4
    Substitute: a,b, = a,b
    a -> goto state 5

State:5
    b -> goto state 6

State:6
    delta -> goto state 1
```

```
State:1
    a -> goto state 2

State:2
    b -> goto state 3

State:3
    STOP

End_Automaton Process_0
```

5.5.8 Disabling

Again, the translation of a disabling expression:

$$B_1[> B_2$$

depends on whether the expression is irregular. An irregular expression will cause the translator to produce an automaton similar to the automaton produced for the enabling expression. The difference is that this time one process may terminate the other and take over control instead of waiting for the other to exit. In fact, if the other process manages to exit the waiting process "disappears" without instantiating itself.

The automaton for an irregular disabling expression can be defined as:

$$M([>) =< \{s_0, s_1\}, \Sigma \cup \{\epsilon\}, \{B_1, B_2\}, C, R, s_0 >$$

The components of the machine are defined as follows:

- C=$\{c\}$ where,
  c =**Dcall** $B_1(\alpha_1, ..., \alpha_n), B_2, (\beta_1, ..., \beta_k)$
- $\Sigma = \{\alpha_1, ..., \alpha_n\} \cup \{\beta_1, ..., \beta_k\}$

- ϵ symbolises the silent move.
- $R = \{(s_0, i, c, s_1)\}$

If the expression is regular then we can construct an automaton by attaching copies of the transitions from the initial state of $M(B_2)$ to each state of $M(B_1)$

If $M(B_1),\ M(B_2)$ are defined as:

$$M(B_1) =< S_1, \Sigma_1, N_1, C_1, R_1, s_{0_1} > \text{and}$$
$$M(B_2) =< S_2, \Sigma_2, N_2, C_2, R_2, s_{0_2} >$$

the automaton produced for the disabling expression can be described as:

$$M(B) =< S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, N_1 \cup N_2, C_1 \cup C_2, R, s_{0_1} >$$

with R being defined as follows:

- $\forall(s_n, g, c, s_k) \in R_1(s), (s_n, g, c, s_k) \in R(s)$,
- For each $(s_{0_2}, g, c, s_k) \in R_2(s)$ and for each $s_i \in S_1$, where $(s_i, \delta, c, s_l) \notin R_1$ then $(s_i, g, c, s_k) \in R(s)$
- $\forall(s_n, g, c, s_k) \in R_2(s) \wedge s_n \neq s_{0_2}, (s_n, g, c, s_k) \in R(s)$,

For example, providing the translator with an input of the form:

```
process process1 [a,b,c] : noexit :=
    a; b;exit [> c;a;b;stop
endproc
```

the finite state machine produced will be of the form:

```
Start Automaton Process_0 (a,b,c )

 State:5
        a -> goto state 6
        c -> goto state 2
```

```
State:6
        b -> goto state 7
        c -> goto state 2

State:7
        delta -> goto state 8
        c -> goto state 2

State:8
        STOP

State:2
        a -> goto state 3

State:3
        b -> goto state 4

State:4
        STOP
End_Automaton Process_0
```

5.5.9 Process Instantiation

Let:

$$P[g_1, ...g_n] := B'$$

be a process definition. The interpretation of a process instantiation of the form

$$P[v_1, ...v_n]$$

is the expression B' presented at the right hand part of the process definition, with gates $g_1, ...g_n$ being renamed as $v_1, ...v_n$.

Therefore, we can translate process instantiations by simulating the moves of $M(B')$ and renaming each gate used in $M(B')$. In other words, if

$$M(B') =< S, \Sigma, N, C, R, s_0 >$$

then

$$M(P[v_1, ..., v_n]) =< S, \Sigma \cup \{g_1, ..., g_n\}, C', R', s_0 >$$

where

- $C' = C \cup \{(g_1 = v_1), ..., (g_n = vn)\}$
- $\forall s \in R, s \neq s_0, R'(s) = R(s)$
  For each $(s_0, g, c, s_k) \in R, (s_0, g, c \cup \{(g_1 = v_1), ..., (g_n = vn)\}, s_k) \in R'(s_0)$

Using this method, tail recursive instantiation is transformed into a conventional loop. Thus, we save the overhead of implementing real process instantiations.

For example the specification:

```
process process1 [a,b] : noexit :=
    a; b; process1[b,a]
endproc
```

will produce the automaton:

```
Start Automaton Process_0 (a,b)

 State:1
        a -> goto state 2

 State:2
```

```
        b -> goto state 3

 State:3
        Substitute:
             a = b
             b = a
        a -> goto state 2
End_Automaton Process_0
```

To achieve this transformation of process instantiations into conventional loops a copy of the initial state is introduced in place of the instantiation. The same procedure is followed even if there is no tail recursion. When the automaton reaches that state it starts simulating the instantiated procedure. This is made clear in the following paradigm:

```
       process process1 [a,b] : noexit :=
                   a; b; process2[a,b]
               endproc
       process process2 [c,d] : noexit :=
                   c;d;stop
               endproc
```

produces the automaton:

```
Start Automaton Process_0 (a,b )

 State:1
        a -> goto state 2

 State:2
        b -> goto state 4
```

```
 State:4
         Substitute:
              c = a
              d = b
         c -> goto state 5

 State:5
         d -> goto state 6

 State:6
         STOP
End_Automaton Process_0
```

5.6 Optimisation of XFSMs

XFSMs created in the way described in the previous section may contain states that can be safely deleted without changing the behaviour of the automaton.

For example, an expression of the form:

```
specification example [a, b] : noexit
behaviour
        process1 [a, b]
where
    process process1 [a,b] : noexit :=
        a; b; process1[a,b]
    endproc
```

will cause the translator to put a copy of the initial state of the automaton that simulates *process1* in place of the instantiation. This will produce the automaton:

```
Start Automaton Process_0 (a,b)
```

```
 State:1
         Substitute:
              a = a
              b = b
         a -> goto state 2

 State:2
         b -> goto state 3

 State:3
         Substitute:
              a = a
              b = b
         a -> goto state 2
End_Automaton Process_0
```

It can be seen that states 1 and 3 have the same transitions. We can eliminate state 3 and transform the previous automaton into the equivalent one:

```
Start Automaton Process_0 (a,b)

 State:1
         a -> goto state 2

 State:2
         b -> goto state 1

End_Automaton Process_0
```

that has exactly the same behaviour.

Another source of optimisation comes from common subexpressions that can be found in a LOTOS specification. For example the expression:

$$a; b; stop[]c; b; stop$$

will cause the translator to produce an automaton for the $a; b; stop$ expression, an automaton for the $c; b; stop$ expression and then a combined automaton made out of the two. This means that the result will look like:

```
Start Automaton Process_0 (a,b,c )

 State:1
         Substitute:
               a = a
               b = b
               c = c
         a -> goto state 6
         Substitute:
               a = a
               b = b
               c = c
         c -> goto state 3

 State:6
         b -> goto state 7

 State:7
         STOP

 State:3
         b -> goto state 4

 State:4
         STOP
```

```
End_Automaton Process_0
```

Again we can eliminate states 3 and 4 and transform the automaton into the equivalent one:

```
Start Automaton Process_0 (a,b,c )

 State:1
         Substitute:
               a = a
               b = b
               c = c
         a -> goto state 6
         Substitute:
               a = a
               b = b
               c = c
         c -> goto state 6

 State:6
         b -> goto state 7

 State:7
         STOP

End_Automaton Process_0
```

The translator uses a hash table to identify *equivalent states*. By the term *equivalent states* we mean states that have the same transitions and only same transitions. Moreover whenever the automaton reaches any of these states, it will always execute the same set of commands.

The key upon states are stored in the hash table is the name of the node of the abstract syntax tree, from which the state was produced. States with the same name are likely to be equivalent. Further more if two states do not have the same name, they can not be equivalent.

Let us take the first example, of process instantiation and consider that at some point process *process1* was instantiated. The node of the abstract syntax tree, where this instantiation was made, will have the name:

$$INST;;INST$$

This consists of:

the name of the root node of the tree that corresponds to *process1*

This name is the string: *;;INST* because we have two action prefixes (*a;b;*) and one instantiation.

the name of the node of the *example* tree.

This is just the string *INST* since the only thing that happens inside the main specification, is the instantiation of process *process1*.

Similarly, the instantiation node of the tree of process *process1* will have the same name. The states produced out of these two nodes are states 1 and 3, and the translator will only check these nodes for equivalence. If the second instantiation had different parameters than the first then the two states would not be considered equivalent due to the different set of commands executed at each state.

By the same method we can detect very fast, equivalent states produced by common subexpressions.

Now, if two nodes have different names then they can not be equivalent because a different name means that there is a different operator somewhere in the path from the node to the leaves of the tree.

Another source of optimisation is the elimination of silent transitions and transitions based on an internal action.

Elimination of silent transitions was discussed previously. Elimination of internal actions is based on the fact that expressions of the form:

$$a; i; B_1$$

have (for an external observer) the same behaviour with the expression

$$a; B_1$$

Therefore, transitions of the form:

$$s_n \xrightarrow{\alpha} s_{n+1} \xrightarrow{i} s_{n+2} \text{ become } s_n \xrightarrow{\alpha} s_{n+2}$$

5.7 Epilogue

In this chapter we presented a method for translating a basic LOTOS specification into a collection of extended finite state machines that are optimised. The work presented here was based in the results of [VSC90, Kar92]. We have assumed that the machines can communicate with each other according to some protocol, which we have not yet described. This is done in the next chapter.

Chapter 6

XFSM Synchronisation

6.1 Prologue

In this chapter we describe a method for synchronising the finite state machines produced by the translator. We explain why the synchronisation of LOTOS processes is so complex, and we propose an algorithm that successfully models the multi-synchronisation feature of LOTOS. We also present the steps must be taken to implement the commands **callP, callE, callD** we introduced in the previous chapter.

6.2 The multi-way Rendezvous

The central idea of a multi-way rendezvous is that some processes are engaged in a situation where a block of statements is executed only when all the processes reach a certain stage of processing.

The multi-way rendezvous, or simply multi-rendezvous, of LOTOS differs from the conventional notion found in other computer languages because of the following uncommon characteristics of LOTOS:

1. The number of processes engaged in a multi-way rendezvous is not known at compile time. This number may vary dynamically during the execution

time, whereas this is not common in other programming languages.

2. A process may be ready for more than one multi-rendezvous at a time. The selection of the wrong one will, possibly, lead to deadlock. For example, consider two processes that are ready for a rendezvous at gates a and b. If one process selects gate a and the other selects gate b, then each of the processes will (infinitely) wait for the other to make an offer. Such deadlock situations must be avoided and we need to find a method where both of the processes select a or b at the same time.

3. A process may have many alternatives at a time. So it may not have to block until a multi-rendezvous is enabled but may follow another path of non-synchronising actions.

In order to solve the above problems we have followed a solution that combines many of the characteristics of other algorithms (e.g [VSC91], [Cha87]) together with the expressiveness of PROMELA language.

6.2.1 The execution model

Because the aim is to translate LOTOS specifications into PROMELA the execution model has to closely follow that of PROMELA's. So, we assume that the run-time model consists of several processes running in parallel and interacting via message channels. We also assume that every LOTOS process has been decomposed into two child processes according to the following schema:

If P is a LOTOS process and B_1, B_2 behaviour expressions then the following specifications give rise to two child processes B_1, B_2 that can interact with their father P via some message channels:

$$P = B_1 >> B_2,\ P = B_1[> B_2,\ P = B_1|[S]|B_2$$

A similar execution model is also presented by Bochmann *et al* in [WvB90].

If we think in terms of XFSMs then the child processes will be generated by the commands:

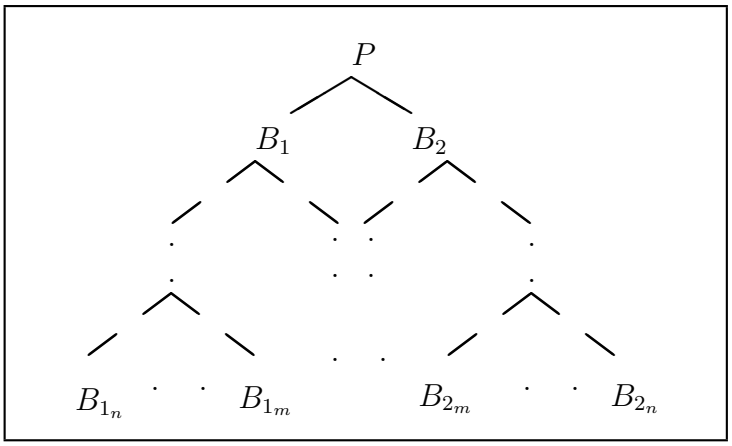

Figure 12: Process decomposition of Lotos specifications

- **callP** $B_1(g_1, ..., g_n), B_2(v_1, ..., v_m), (s_1, ..., s_m)$
- **callD** $B_1(g_1, ..., g_n), B_2(v_1, ..., v_m)$
- **callE** $B_1(g_1, ..., g_n), B_2(v_1, ..., v_m)$

where a process can be regarded as an instantiation of an extended finite state machine.

This decomposition will lead us in a tree structure like the one presented in Figure 12. Each of the leaf processes behaves according to a LOTOS behaviour expression, while the processes that correspond to interior nodes have only control duties.

A run-time process terminates its execution when it reduces to *STOP* or when both of its components finish their execution.

There are three types of LOTOS actions that a run-time process can execute:

1. Internal or hidden actions
2. Actions that must be synchronised with the environment
3. Actions that must be synchronised with other processes

Although a process does not know the exact number of participants in a multi-rendezvous, it knows the category in which an action falls. Therefore, our solution is based on this fact.

We assume that every run-time process randomly selects one rendezvous for which it is ready. If the rendezvous falls into one of the first two categories then the process needs no kind of synchronisation so it can proceed after performing the necessary operations. On the other hand, if a process selects a multi-rendezvous action then it has to inform its father of the selection and wait for an answer from him. The father will check whether the rendezvous is enabled at the time and it will eventually answer with a *PROCEED* or *RETRY* message.

This *select-check-select again* model will handle properly all the correct specifications but it will probably cycle for ever in case of an incorrect specification. This is because none of the possible actions will be enabled and the child will always receive a *retry* message. PROMELA's verification statements are used in such cases in order to identify these livelock situations.

6.2.2 The Algorithm

Assumptions and Definitions

We assume that each run-time process is connected with its father via two message channels. More precisely, a child can send to its father one of the following messages, via the use of the message channels **rndzvous, ansr**:

rndzvous[*gate name*, *list of events*]

ansr[*answer*]

where **rndzvous** is a message channel that can hold one message per time and **ansr** is a synchronising message channel.

The *gate name* may be:

- either a special *NOSYNC* message, either
- a *NOT_ESTABLISHED* message, or

- a name denoting a gate of the original LOTOS specification.

The *list of events* may be either *EMPTY* or the actual list of events offered at the gate *'gate name'* of the LOTOS specification. For this translator the list of events is always *EMPTY*, since only basic LOTOS is considered. The *answer* may be either *PROCEED*, or *EXIT* or *RETRY*.

The way channels are used as well as the meaning of each message will be explained later.

We also define the *synchronisation set* of a run-time process to be the set of gates on which it must synchronise with other run-time processes. This set, since it varies at execution time, is passed as a parameter from a father process to its child.

We say that a gate is synchronising with other processes (multi-synchronising) if it belongs to the synchronisation set of the process. We say that a gate is non-synchronising if is does not belong to the synchronisation set of the process.

We also use two more sets:

The *control set* contains the gates where the two child processes must be synchronised. It can be determined statically. The *highest-control* set contains the gates on which the children of a process, but not the process itself, must be synchronised.

Formally is defined as:

$$highest\ control\ set\ =\ control\ set\ -\ (control\ set\ \bigcap\ synchronization\ set)$$

A process can decide whether a rendezvous at a gate may take place only when this gate belongs to the highest control set (of the process).

Description of the Algorithm

In general the algorithm can be described as follows:

Each process locally and randomly selects a rendezvous and informs its father of the selection. This information travels upwards the tree (each process informs its father process and so on) and eventually reaches the root of the tree.

As soon as the information is transmitted the process is blocked waiting for an answer from its father. Each father process can either command his children to *proceed* or to *retry*. The answer depends on the gate at which the rendezvous is going to take place as well as on the availability of the other child for a rendezvous.

More precisely, if the selected gate is an internal gate or a non - synchronising gate, the process issues a

rndzvous(*NOSYNC*,*EMPTY*) message

and then blocks, waiting for an acknowledgement answer from the father.

On the other hand, if the selected gate must synchronise with another process, the father is informed to check the availability of the other child, by a

rndzvous(*gate name*, *list of events*) message

and the process blocks as above.

Eventually, the father receives messages from both of his children. Then he checks whether a rendezvous is enabled. The conditions for a multi-rendezvous to take place are described in [ISO89a]. It is obvious that if a

rndzvous(*NOSYNC*,*EMPTY*) message

has been sent, no multi-rendezvous is enabled.

If there is no multi-rendezvous possibility then the father issues a

rndzvous(*NOT_ESTABLISHED*,*EMPTY*) message

to the grand-father process and blocks waiting for an acknowledgement. This goes on, till we reach the root node of the tree.

If the rendezvous is enabled, we have to remember that a father is authorised to send a *PROCEED* answer in a multi-rendezvous query only if he is the highest controller of the gate at which the multi-rendezvous is going to happen.

If he is not, he has to inform his own father that he wants to participate in a multi-rendezvous by issuing a **rndzvous**(*gate name*, *list of events*) message and entering a blocked state, where he waits for the answer.

A higher controller will follow two steps. The first step is to decide whether the multi-rendezvous is enabled and (provided that it is) the second step is to inform its father via a **rndzvous**(*NOSYNC*, *EMPTY*) message and move into a blocked state waiting for an answer. A **rndzvous**(*NOSYNC*, *EMPTY*) message informs the nodes of the tree that the branch is not available at this time for a rendezvous and they should reject any request for multi-synchronisation made by the mirror branch of the tree.

When the **rndzvous**(*NOSYNC*, *EMPTY*) reaches the root node then this node will answer by issuing a *PROCEED* answer. This answer will propagate downwards the tree and it will instruct each node to proceed in the actions decided at the first step of their calculation.

These actions may be either the transmission of a *PROCEED* answer to their children or the transmission of a *RETRY* answer.

Whenever a leaf process receives a *PROCEED* answer, it proceeds to the next action, implied by the specification, whereas if it receives a *RETRY* answer it makes another selection out of the possible ones.

The father will send his answer (*PROCEED* or *RETRY*) via the **ansr** channel, unblocking the child process that listens to that channel waiting for the answer.

This algorithm will work as long as the initial specification contains no deadlock situations. However, since verification of protocols is the aim, we need a method for handling deadlock situations. In case of a deadlock this algorithm will force each of the child processes to cycle through **rndzvous** and *RETRY* messages. That means that deadlocks will be transformed into livelocks in PROMELA language. By using the **accept** statements we can identify such livelock situations.

An example

Let us consider the following specification:

```
process P[g,l]:noexit := D[g,l] | [g] | g;A[g,l] endproc
```

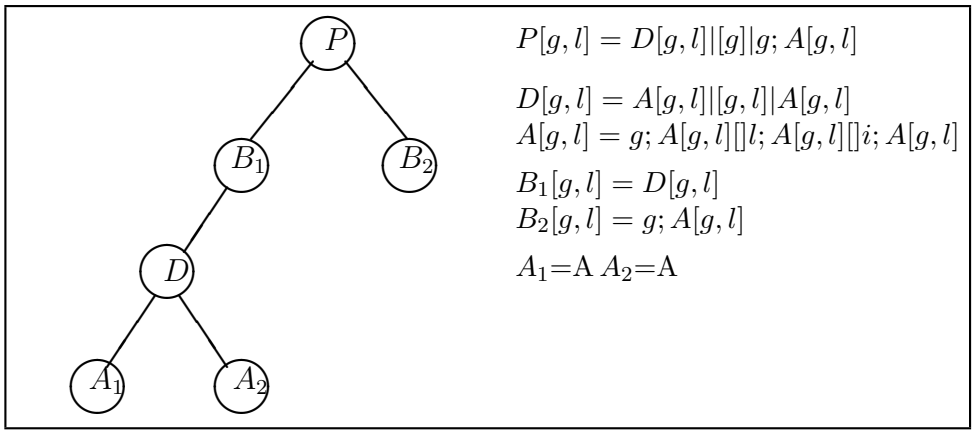

Figure 13: A specification and the resulted execution tree.

```
process D[g,l](noexit):= A[g,l] | [g,l] | A[g,l] endproc
process A[g,l](noexit):= g;A[g,l] [] l;A[g,l]
                                  [] i;A[g,l] endproc
```

This specification is written in basic LOTOS, so no *list of events* appears while each action is denoted, solely, by a gate name.

The specification will lead to a decomposition, like the one presented in Figure 13.

A possible scenario is the following one:

First, A_1 and A_2 select i and g gate events respectively, while B_2 (naturally) selects g. In this case, according to the algorithm described above, the following events will take place:

All the processes will inform their father and they will block waiting for an answer. So, B_2 sends a **rndzvous**('g') message, A_2 a **rndzvous**('g') message while A_1 sends a **rndzvous**(*NOSYNC*) message. After this, all will block, listening to the **ansr** channel for an answer.

Eventually, process D will collect the two messages issued through the **rndzvous** channel from processes A_1 and A_2 and it will conclude that the rendezvous at gate g cannot take place, since one of the children decided to proceed on his own.

So, it will send a **rndzvous**(*NOSYNC*) message to its father (process B_1) and it will block. Process B_1 will collect the message and it will retransmit it to process P. Eventually, process P will collect two messages:

One from process B_1 and the other from process B_2.

P is the root node of the tree, and it is authorised to issue answers. It is obvious (by the **rndzvous**() messages) that the rendezvous at gate g cannot take place, so it issues an **ansr**(*RETRY*) answer to process B_2 and an **ansr**(*PROCEED*) answer to process B_1.

As soon as B_2 receives the answer it will make another choice. Here, since it has no alternatives, it will choose again the rendezvous at gate g.

The other answer will travel downwards the tree and it will reach process D. Process D, will proceed then to the actions that had been decided before the process was blocked. These actions are the transmission of an **ansr**(*RETRY*) answer to the A_2 process and the transmission of an **ansr**(*PROCEED*) answer to the A_1 process.

Finally, the leaf processes (A_1 and A_2) will receive the answers. A_1 will proceed to the instantiation of another A process, while the other one will try to make another selection out of the possible ones (i.e an action at gate g, gate l or the unobservable action i).

A different route of events could be the following:

Both the A_1 and A_2 processes choose to establish a rendezvous at gate g. So they both send a **rndzvous**('g') message and then they move into a blocked state waiting for an answer.

Process D, collects the messages and tries to establish the rendezvous. Since both of the processes agree on the rendezvous but gate g does not belong to the highest control set of D, D issues an **rndzvous**('g') message and moves to a blocked state, too.

Eventually process P will collect two **rndzvous**('g') messages and it will send a *PROCEED* answer to both of its children. This answer will propagate downwards and after a while, it will reach all the leaf processes commanding them to proceed to the next action.

6.3 Implementation of callP

The multi-rendezvous algorithm actually implements the callP command. In general, this command can be implemented as follows:

A command of the form:

$$\textbf{callP } B_1(g_1, ..., g_n)B_2(v_1, ..., v_m)(s_1, ..., s_k) \ )$$

generates two processes: (B_1, B_2).

Since we know the gates of B_1 and B_2 at compile time, we only need to change their type at execution time. So, if a gate belongs to the set $(s_1, ..., s_k)$, we change its type into *SYNC_SET* and we pass it as a parameter to B_1 and B_2. Any gate appearing in a *hide* list is considered to be an internal gate and its type is *INTERNAL*. Finally, any other gate is considered to be synchronised with the environment. We denote this by the *SYNC_E* type.

After that, the behaviour of B_1 and B_2 is defined by the multi-rendezvous algorithm.

Every time a child process stops, the father is informed by a *STOP* message to consider only offers made by the other child. When both of the children have stopped, the father himself issues a *STOP* message to denote that he is also exiting.

An *acceptance label* is used at the stage where the father of B_1 and B_2 issues a *NOT_ESTABLISHED* message.

We have explained earlier that deadlock situations are transformed into livelock situations in PROMELA. By using this label at that stage, we state that it is not possible for a father to continualy transmit *NOT_ESTABLISHED* messages

since this implies that the protocol can not proceed further. Since a deadlock in LOTOS can only be produced as a result of a parallel operator, the only point where *acceptance labels* are used, is here.

The code produced each time the parallel operator is met can be found in Appendix B.

6.4 The callD command

The algorithm that carries out the callD command, is closely connected with the schema implied by the multi-rendezvous algorithm. The general idea can be described as follows:

Whenever a

$$\textbf{callD } B_1(g_1, ..., g_n)B_2(v_1, ..., v_m) \ ) \ )$$

command is found, B_1 process is instantiated. The B process (which is the father of B_1) receives messages from its child and retransmits them to its own father. The crucial point is that whenever a message is received from the B_1 process, (through the **rndzvous** channel) a non-deterministic choice is made between:

1. the first action of B_2 or,
2. the propagation and further process of the **rndzvous** message.

If we choose the first route then we issue a special **ansr(***EXIT***)** message to the B_1 process and we continue with B_2, otherwise B_2 is not instantiated and we continue to handle messages of B_1.

An *EXIT* message causes a process (and the children of the process) to exit. If B_1 process is reduced to *STOP*, then B exits even if B_2 has not been instantiated.

Appendix C contains the PROMELA code for the disable operator.

6.5 The callE command

The **callE** command is carried out in a similar way.

Whenever a

$$\textbf{callE } B_1(g_1, ..., g_n)B_2(v_1, ..., v_m)$$

command is found, B_1 process is instantiated. After this, each message transmitted to process B is retransmitted to the father of B.

Process B checks each time if an action at gate δ is offered. Such an action means that process B_1 exits successfully. So, whenever this offer is made, process B_2 is instantiated as soon as the B_1 process exits and the behaviour of B_2 becomes the behaviour of B.

The PROMELA code for the enable operator can be found in Appendix D.

6.6 Epilogue

In this chapter we have presented a method for synchronising LOTOS processes. The algorithm was designed with three main purposes:

- To be applicable to the model of XFSMs produced by the translator
- To be transformable to PROMELA language
- To produce a minimum interleaving of processes

This algorithm has no actual limit to the number of processes that can handle. It should be noted here that although the algorithm presented in [VSC91] has a better overall performance by sending upwards all the event offers available at a time (and not one by one), it is not followed here mainly because the resulted PROMELA code would be too complicated. The handling of sets of event offers would require:

1. *larger communication channels*

In order to send all the possible rendezvous a process is ready to participate in, all at one time, it would be necessary to create channels that can facilitate more than one message at a time. So, during compile time we would have to find out the maximum number of rendezvous that a process participates in and, later, produce channels that can held up to that number of messages even if the process does not finally joins all of the rendezvous. This is because we can not change the capacity of a channel dynamically during execution time. Thus the complexity of the resulted code would be much higher, since the larger the channels are, the harder is to verify the protocol [JH91].

2. *Complicated target code*

   More over, the code needed for handling set operations would be messy and complicated mainly because PROMELA does not support set operations. So, this code would add extra unnecessary states to the resulted validation model.

In the following chapter we will propose a different algorithm for the synchronisation of processes that works only for static models. However, both of them will fail to produce an extended analyser for LOTOS specifications. The reasons will be explained in the next chapter.

Chapter 7

A different algorithm and a discussion

7.1 Prologue

In this chapter we will present another method for synchronising finite state machines. Unlike the method introduced in Chapter 6, this algorithm puts an upper limit to the number of processes that can be handled, but it uses a minimum number of message exchanges.

We will also show that even with a simple model like this one, the number of states produced is too high for an exhaustive or even a partial analysis.

7.2 The execution model

Again, we assume that the execution model consists of processes running in parallel, decomposed due to **callP, callD** and **callE** commands and interacting via message channels. We also assume the existence of a global process table, as well as the existence of several local ones.

Finally, we categorise LOTOS's actions into the same sets of

- synchronising with other processes

- synchronising with the environment
- synchronising with none (internal actions)

The difference in this algorithm is that processes inform the highest controller of each gate directly, instead of informing always their direct father. They also inform all the highest controllers at once, instead of using a *select and try* technique.

7.3 Description of the Algorithm

7.3.1 Assumptions and definitions

We assume the existence of a global process table:

$$\textbf{global\_proc\_table}[\mathrm{K}]$$

that can hold up to K process identifiers.

We also consider the existence of two message channels:

$$g_i\_req(message\_type, process\_id, index\_entry)$$
$$g_i\_conf(message\_type, new\_proc\_id)$$

for each gate g_i defined in the specification.

Moreover we assume that each process can communicate with its father via a special message channel:

$$\textbf{father}(message)$$

The meaning of each parameter passed to the channel will be explained later.

In addition, we use a number of local process tables:

$$\textbf{local\_proc\_table}[\mathrm{N}][\mathrm{N}]$$

where $N < K$.

We assume that each time a process B generates two child processes B_1, B_2 that must synchronise at gates $g_1, ..., g_n$, n more processes are created. Each of these processes control one of the synchronising gates. We will refer to them as *ports* [PS92], and we will associate with each one of them a gate.

When processes B_1, B_2 are created they communicate with each port g_i via the pair of channels g_i_req and g_i_conf.

In addition, each one of the ports knows the number of participants needed for a rendezvous to take place at the gate they control.

Processes, that play the role of participants in a rendezvous, are identified by unique process identifiers. We will explain later how a port can be informed for the number and the id's of the participants.

7.3.2 The algorithm

The algorithm consists of two parts:

The first part deals with the processes that want to participate in a rendezvous. Processes send via channel g_i_req a *READY*(idx, pid) message, to each one of the ports that control the gates the process is ready to offer an event, where *pid* is the process identifier that can be found at the *idx* entry of the symbol table **global_proc_table**.

Then the process blocks, waiting for a (*PROCEED*, newpid) message at one of the g_i_conf channels.

The second part of the algorithm deals with the collection of *READY* messages.

Each port listens to the channel g_i_req and collects requests for a rendezvous. Each time a process with process id *pid* sends a request message, the port stores the *pid* and *idx* to its local symbol table **local_proc_table**.

When the port determines that a sufficient number of *READY* messages has been collected, it tries to lock all the processes that submitted these messages. To prevent deadlocks, processes are locked in the order defined by their entry at

the symbol table.

To lock a process with pid *pid* whose index at the symbol table is *idx*, ports look at the global symbol table entry *idx*. If the number stored there is equal to *pid* then the process is locked by changing that number into *-pid*. To make sure that no one else has managed to lock the process in the time between the *reading* and the *updating* of the global symbol table we can make use of an *atomic* sequence. If the number stored there is equal to *-pid* then another port has managed to lock the process. In this case, we have to wait till one of the following happens:

If the entry at the global symbol table changes to *pid* then the process is unlocked again and we can try to lock it.

If the entry is neither *pid* nor *-pid* the process has participated at another rendezvous at is no longer available. Then the port unlocks all the processes it locked and returns to the stage of collecting messages.

If the port manages to lock all the processes needed for a rendezvous then

1. it changes the entry of each process at the global symbol table to a new number, greater than the previous one. This is the *new* identifier number of the process. This new pid is the same for all processes that participated at the rendezvous.

2. It issues a

$$g_i\_req!PROCEED(newpid)$$

   where *newpid* is the new pid of the process.

3. It returns to the stage of collecting *READY* messages.

Finally, each of the processes that participated to a successful rendezvous use the new process id number assigned by the port where the rendezvous took place, the next time they issue a *READY* message.

This algorithm will work as long as the number of processes participating in a rendezvous is constant. If this number changes due to the generation of two new processes, then we can inform each port by following the next schema:

Right before a process spawns two new children, it informs its father for the generation. This is done via the use of the **father** channel. Then the process is blocked waiting for an answer from him. The father then informs each of the interested ports to increase the number of participants needed for a rendezvous, informs his own father and blocks.

Eventually, all the nodes of the branch of the tree that contained the leaf that is about to generate new processes will be informed. When the message reaches the top, the root node will issue another message that this time will travel downwards and will unblock all the processes in the branch, enabling the birth of the new children.

7.4 Discussion of the algorithm

Although there are algorithms ([Gv89]) that handle the dynamic creation of processes in a more efficient way, these can not be used in PROMELA. This is because they assume the presence of a dynamic structure than can be easily extended. In the case of [Gv89] the presence of a ring (representing a gate) is assumed, on which all the processes that participate in the same rendezvous are connected. If one of the processes generates new ones, it just extends the ring by adding a pointer to the newly created processes.

In PROMELA where dynamic data structures (in this case the ring) are not permitted we can not use this method. Thus, although the pattern suggested earlier, where each node of the tree informs its predecessor, may seem time-consuming it is the only one that can be implemented in PROMELA. Also, due to the lack of dynamic structures the process tables used are static which means that there is an upper limit to the number of processes that can be handled.

However, the latter is not a major problem. The size of the process table can

be easily changed to handle as many processes as we want, through the use of a `#define` statement in the PROMELA code.

Moreover, the systems we usually want to verify have a predefined size. For example, a computer system has a predefined CPU and memory capacity. Dynamic structures try to implement the notion of *infinity* in computer systems. But in real systems *infinity* is unimplemented. A buffer may be specified as being infinite but we will always have to put an upper limit because when it comes to reality there can be no buffer with an unlimited capacity. Therefore, we can not consider that as a drawback.

In addition, this algorithm uses a minimum number of message exchanges. For the common case it takes two message transfers for a rendezvous to take place. Of course, the rare case (process generation) adds an overhead of $2(n+k)$ message transfers where n is the depth of the process tree and k the number of ports but we can greatly minimise that by writing specifications with a minimum number of dynamic process creation.

However, even if we use this algorithm we can not fully analyse even small protocols.

Suppose that we have a process like the one depicted in Figure 14.

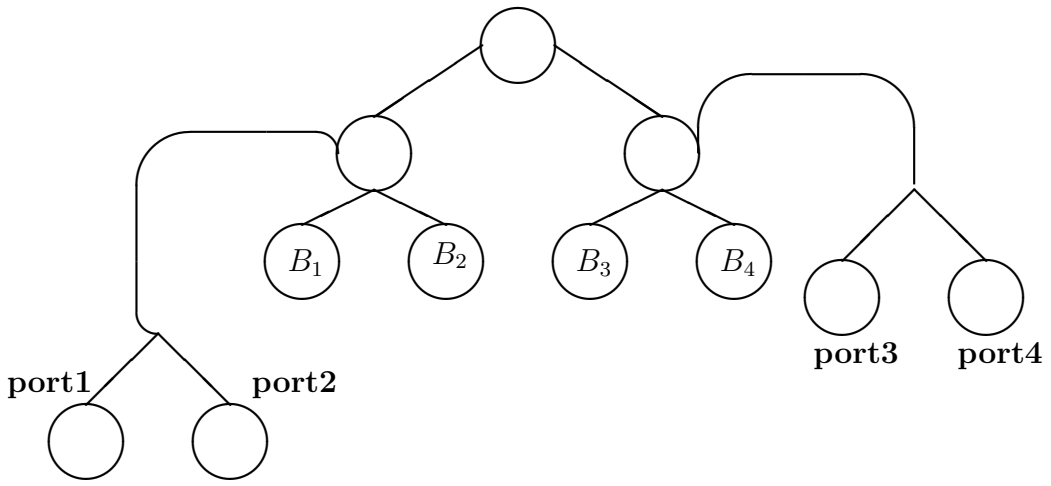

Figure 14: Instance of a process tree

There are four active process B_1 to B_4 and four active ports. We assume that processes B_1 and B_2 synchronise at ports one and two, and they are ready to offer an event to any of the two. We also assume the same for the mirror processes B_3, B_4 and ports.

Finally we consider the case where ports do not have to continuously listen to the **father** channel for a change in the number of participants at a rendezvous, i.e we are considering a static model.

From the discussion in the previous section we can easily deduce that an automaton simulating **port1** may be in one of at least thirteen different states. This is because there are four different states where the port is receiving or sending messages. Four different states used by the port to store, or update variables that have to do with the messages received or sent. There are two different states that perform the locking procedure for each of the processes. Finally there are two states where the automaton waits in case where a process is currently locked by the other port and one state where the port moves when it has to abandon the locking procedure.

Consequently there is an automaton with at least thirteen different states for each port. Additionally, we have an automaton with four different states for each of the active processes. Two states where the *READY* messages are sent to each of the ports and two states where the process collects either one of the *PROCEED* replies.

In order to verify the above schema, we have to analyse all the possible execution sequences.

Nevertheless there are of course some impossible sequences. For example it is impossible for ports **port1** and **port2** to send a *PROCEED* reply both at the same time. In fact, the only impossible combinations are those that have **port1** and **port2** at the stage of submitting *PROCEED* answers and those where both of the ports abandon the locking procedure. We have stated before that it takes four states for a port to submit a *PROCEED* answer to each of the two processes.

Therefore the possible combinations for the left part of the tree produce:

$$Number\ of\ states = 2 \times [(13 - 5) \times 13] \times 4 \times 4 = 3328 \quad (1)$$

The left part is fully interleaved with the right part. So the total number of states is:

$$Total\ number\ states = 3328 \times 3328 = 11075584 \quad (2)$$

It is clear that the number of states the validator has to analyse is too high for a simple paradigm like this one.

Therefore, full searching of real LOTOS specifications that may contain up to ten or eleven running processes and twenty different ports is rather impossible.

7.5 Epilogue

In this section we have presented an algorithm for multi-synchronisation where there is a limit on the number of process instantiations but a minimum number of message transfers.

However, even with this simple model we can not effectively verify network protocols. The problem rises from the fact that each time we have to verify a protocol written in LOTOS, we also have to verify the synchronisation algorithm. This adds a tremendous overhead to the already large problem of validating the protocol itself.

Clearly, the case would be different if PROMELA had the means for implementing multi-synchronisation.

Chapter 8

The data part of LOTOS

8.1 Prologue

Anyone, with even a little experience in LOTOS can observe that LOTOS specifications consist of two components: the data and the control component, that naturally divide the process of translating LOTOS into two parts.

This translator does not consider the data part at all. In this chapter we will describe the method followed for translating the data part and we will explain why it is not possible to have a translator from full LOTOS to PROMELA.

8.2 Translation of the data part

The main function of a data translator will be the provision of a procedure to decide equivalence of value expressions.

LOTOS data part provides the translator with a set of *terms*, *operations* and *equations*. Based on this information the translator should be able to prove whether two terms are equal or not.

Consider the (full) LOTOS expression:

$$[E_1 = E_2] \longrightarrow B \tag{3}$$

The translator for the control part should know if the expressions E_1, E_2

represent the same value, otherwise it is not able to proceed to the execution of B.

The proof of whether E_1 and E_2 are equal is not trivial. Remember that in LOTOS the user specifies his own data. In a typical programming language, an expression like (3) is evaluated based on operations and equations already known to the compiler.

For example if $E_1 = 1 + 2$ and $E_2 = 3$, the compiler will calculate the value of $(1 + 2)$ based on the standard '+' operation and it will conclude that the two expressions are equal.

However, in LOTOS the '+' operator may have a different meaning. In fact, the translation of '+' changes according to the definitions in the data part. For this reason, there is a need for a different data compiler that will be able to prove the equivalence of two expressions based on the definitions made in the specification.

In the following we will try to informally explain what is considered to be equivalence of terms in LOTOS.

The data part of LOTOS defines an *algebra*. An *algebra* can be thought as being a set of elements together with some operations defined on them.

Named set of elements are called *sorts*. In LOTOS, operations take elements of one sort and map them into an element of (possibly) another sort. The elements of a sort are called *ground terms*. The *signature* of an algebra consists of a set of *sorts* and *operations*. A *signature* defines the elements of the *algebra*. Any element of the *algebra* can be constructed by the operations defined in the *signature*, if these *operations* are applied upon the elements of the *sorts*.

Equations come to categorise the elements of an *algebra* into equivalence classes. As one would expect an *algebra* is fully defined if we know its *signature* and its *equations*.

Two elements a, b are considered to be equal if they belong to the same equivalence class. That means that a series of equations must exist such that if

followed they will lead to a statement of the form:

$$a = b \tag{4}$$

To achieve this, we have, for example, to start by an equation having a at the left hand, and continuously substituting the left hand of the equation with the right part. Fortunately, we will reach a stage where either $a = b$ or further substitutions are not possible.

However, there is always the danger to cycle forever through substitutions without reaching a result. To avoid such infinite behaviour and to be possible for automated tools (e.g. data compilers) to parse them, equations of LOTOS should have the *finite termination* property [BA89].

This process of substitution is known as *rewriting*. In rewriting we view each equation as being true only from left to right. The equation states that the term at the left of the equation can be rewritten like the term at the right part of the equation. If there is a series of transformations where t_1 can be rewritten as t_2, t_2 as t_3 and so on until t_n then

$$t_1 \equiv t_n \tag{5}$$

which should be read as t_1 being equivalent to t_n.

Most of the compilers that handle the data part of LOTOS implement such a rewrite system.

Whenever the proof of equivalence between two terms is needed, the compiler for the control part calls an *evaluate* function that implements the rewrite system. Eventually, the function returns a **true** or **false** answer to the compiler.

Unfortunately this kind of interaction is not possible in PROMELA. We will explain the reason in the following section.

For the moment we will not expand further to the data part of LOTOS. The reader can find more information into [BA89, AMn88]. In [BA89] a description of an abstract data type interpreter is made and the process of producing *rewriting* rules is mentioned. A tutorial for algebras and abstract data types, intended for computer scientists and not mathematicians can be found in [AMn88].

8.3 The data part in PROMELA

The previous discussion made clear that it is not wise to produce PROMELA code for evaluating expressions. PROMELA is a language intended for specification and validation and does not have the flexibility of real programming languages like C, for example.

A rewrite system does not need any of the constructs PROMELA provides. We do not need neither synchronisation nor non-determinism. Temporal claims and *accept* statements would be of little use. More over, the rewrite system would add a tremendous overhead to the verification algorithms.

Consequently, the obvious solution is to combine some C functions together with the PROMELA code. Whenever the evaluation of two expressions is needed, a C function called *eval* will be called. The validator will continue execution according to the result returned by the function *eval*.

However, PROMELA does not permit the use of C functions, except macro definitions. Even if we tried to extend PROMELA with C calls that run independently of the validator, we would have major problems.

The reason is obvious. Suppose that at some stage the validator has to call *eval* to decide upon the equality of two expressions. In Figure 15 we represent this situation:

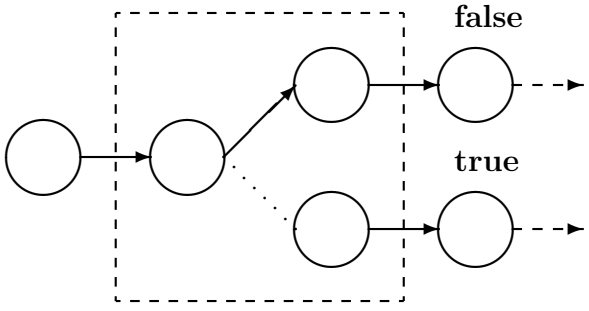

Figure 15: Incorporating a data compiler

In this figure, the data compiler is represented as an automaton, surrounded

by dashed lines. The transition represented by dots is an impossible transition.

The states outside the dashed lined rectangular represent states of the translator.

Since a validator tries to analyse all possible execution sequences , it has to test the specification for both a **true** and a **false** answer. But the figure shows that a **true** answer is impossible. The validator does not know that, since the data compiler is invisible to it. Therefore, when the validator tries to analyse the sequence generated by a **true** answer it will probably report unreliable results.

Actually, SPIN and PROMELA do not support the inclusion of calls to C procedures because the validator would loose track of everything that happens inside the unknown C procedure and would not be able to back up in the depth-first search after a procedure had been invoked in forward steps. It would also allow users to hide static data inside procedures - that would destroy the working of the validator all together because these data structures would encode state information which should become part of the state vector to keep the verification correct. If these structures are not included in the vector, the validator has no means of producing a reliable answer.

For the reasons explained above the inclusion of a data compiler that will work together with the control part was not possible. That is why we have restricted the input to be only the control part of LOTOS. A simulator or a validator for the full part of LOTOS must be written in such a way that will have access to all the parts of the specification. Unfortunately, we could not do that in PROMELA.

8.4 Epilogue

In this section, we briefly discussed the steps that must be followed to produce a compiler or interpreter for the data part. We have also explain why it was not possible to take this steps at this case and restrict the range of specifications that can be translated.

But although it is not always possible to translate full LOTOS specifications

there is a class of protocols that can be parsed by the translator if they are first translated into basic LOTOS.

A behaviour oriented specification, i.e a specification that contains no predicates, written in full LOTOS can be transformed into basic LOTOS if we create a gate name for each possible value involved in any value offer in the specification. Then we can map each matching event to a pure event at one of the new gate names. Of course, only values taken from a finite set can be mapped [MV92]. Such kind of transformations can be easily automated, and there are tools that perform them.

Chapter 9

Conclusion

9.1 Prologue

This chapter contains information on the state of the translator and provides some guidelines for future research.

The first section refers to what has been implemented and what has left for future extensions of the project.

The second section addresses issues that are currently under research and problems that are still unsolved.

9.2 State of the Translator

At this moment the translator implements the following:

- It accepts as input a basic LOTOS specification and produces abstract syntax trees.
- It searches the abstract syntax trees and correctly identifies irregular expressions.
- It produces finite state machines out of the annotated abstract syntax trees.
- Optimises the finite state machines produced

- Generates PROMELA code by implementing the (dynamic) multi rendezvous algorithm presented in Chapter 6.

Work has been left for the syntax analysis phase where the production of user friendly error messages can be added. The creation of finite state machines can be extended so that first actions are identified. Finally, the code generator can be extended by including the algorithm presented in chapter 7 and improving the code produced for the other algorithm.

However, as the results of this project show, no matter what the improvements will be, a translator from LOTOS to PROMELA can not be used for extended analysis of the behaviour of protocols. The main difficulty appears to be the multi-synchronisation feature of LOTOS, a problem that although it was present from the start, it became apparent only during the later stages.

9.3 Further Work

Although the previous guidelines will extend the translator they will not solve the problem of the large number of states produced. One solution may be to follow the idea presented in [CHJ91]. There, the process of finding out the participants of a rendezvous is done during compiling. So, this overhead is removed from the verification procedure. Of course, that also means that the compiler has to do a lot of *simulating* in order to identify which processes generate new ones and which rendezvous are affected by that. However, it is not clear to what extend is possible to perform an exhaustive simulation of the specification during compiling in order to find out the participants of a rendezvous in every feasible execution sequence.

In any case, random simulation is still possible. For this reason the translator could be extended in such a way that guided simulation will become feasible. For example, the user may specify which XFSMs should interact each time forcing the translator to produce code for only that XFSMs.

Producing XFSMs that are not bind with the original specifications with strong equivalences and allow for further optimisations may be possible.

On the other hand, the translator can easily produce executable code if combined with a data compiler. To achieve that we can use the XFSMs produced, to derive code written in another language, like C for example.

The data part of LOTOS has proved to be too cumbersome to be implemented. Research has been going on to find a more suitable data part for LOTOS.

Finally, we should mention that there are a lot of open problems in the area of prototyping formal description techniques and analysing large specifications written in any of these languages.

It seems that with the current knowledge and technology we are not able to effectively verify computer protocols.

9.4 Epilogue

In this thesis we have described in detail the construction of a translator from LOTOS to PROMELA. We have presented the basic constructs of LOTOS and PROMELA and we have described the SPIN validator. We have shown that only the control part of LOTOS can be implemented into PROMELA. We have carry out the translation by producing XFSMs that communicate via a multi synchronisation algorithm.

Finally, we have presented two synchronisation algorithms, and we have shown that only random simulation is possible for translated LOTOS specifications.

Appendix A

Basic LOTOS

A.1 Prologue

This appendix describes the context free grammar of basic LOTOS. It is presented in a form accepted by the parser *Yacc*.

A.2 The grammar

```
%token SPECIFICATION
%token BEHAVIOUR
%token END_SPECIFICATION
%token WHERE
%token PROCESS
%token DEFINITION
%token END_PROCESS
%token EXIT
%token NOEXIT
%token IN
%token FAT_BAR
%token SYNCHRONIZATION, INTERLEAVING, LEFT_PAR, RIGHT_PAR
```

```
%token HIDE
%token ENABLE
%token ACCEPT
%token DISABLE
%token INTERNAL_EVENT %token STOP
%token IDENTIFIER

%start module

module:          specification
        ;

specification  : SPECIFICATION
        specification_identifier formal_parameter_list
        BEHAVIOUR
        spec_definition_block
        END_SPECIFICATION
        ;

I1option:         /* empty */
        |process_definitions
        ;

spec_definition_block:  behaviour_expression I1option
        ;

definition_block:       behaviour_expression
        ;
```

```
I2repeat:               process_definition
        |I2repeat process_definition
        ;

process_definitions:WHERE I2repeat
        ;

process_definition     : PROCESS
          process_identifier formal_parameter_list
          DEFINITION
          definition_block
          END_PROCESS
        ;

I30option:         /* empty */
         | gate_parameter_list
        ;

formal_parameter_list:   I30option functionality_list
        ;

gate_parameter_list:    '[' gate_declaration_list ']'
        ;

functionality_list:     ':' EXIT
         | ':' NOEXIT
        ;

behaviour_expression:   hiding_expression
```

```
         | enable_exp
        ;

parallel_operator:      SYNCHRONIZATION
         | INTERLEAVING
         | LEFT_PAR I35option RIGHT_PAR
        ;

I35option:         /* empty */
         | gate_list
        ;

hiding_expression:      hiding_operator behaviour_expression
        ;

hiding_operator:        HIDE  gate_declaration_list IN
        ;

enable_exp:             disable_exp
         | enable_expression
        ;

enable_expression:      disable_exp enable_operator enable_exp
        ;

enable_operator:        ENABLE
        ;

disable_exp:            parallel_exp
```

```
         | disable_expression
        ;

disable_expression:     parallel_exp DISABLE disable_exp
        ;

parallel_exp:           choice_exp
        | parallel_expression
        ;

parallel_expression:     choice_exp parallel_operator
                         parallel_exp
        ;

choice_exp:               guarded_exp
        | choice_expression
        ;

choice_expression:        guarded_exp FAT_BAR choice_exp
        ;

guarded_exp:              action_prfix
        ;

action_prfix:             action_prefix_expression
         | atomic_expression
         | process_instantiation
        ;
```

```
action_prefix_expression: action_denotation ';' action_prfix
        ;

action_denotation:        gate_identifier
         | INTERNAL_EVENT
        ;

atomic_expression:        STOP
         |  EXIT
         | '(' behaviour_expression ')'
        ;

I41option:         /* empty */
         | actual_gate_list
        ;

process_instantiation:   process_identifier I41option
        ;

actual_gate_list:        '[' gate_list ']'
        ;

I49repeat_sep:            gate_identifier
         | I49repeat_sep ',' gate_identifier
        ;

gate_list:                I49repeat_sep
        ;
```

```
I50repeat_sep:            gate_identifier
         | I50repeat_sep ',' gate_identifier
        ;

gate_declaration_list:    I50repeat_sep
        ;

specification_identifier: IDENTIFIER
        ;

process_identifier:       IDENTIFIER
        ;

gate_identifier:          IDENTIFIER
        ;
```

Appendix B

The callP command

B.1 Prologue

Each time an **callP** command is met the code presented in the next section is produced.

HCS denotes the *highest control set* of a process.

$< gate, gatetype >$ is the list of gates with their types that a process has access to.

B.2 The code

```
proctype B(<gate,gate_type>;chan rndzvous;chan blck)
{
        chan rndzvousL=[1] of {int};
        chan rndzvousR=[1] of {int};
        chan blckL=[0] of {int};
        chan blckR=[0] of {int};
        bool LEFT,RIGHT;
        int message;
        int gateL,gateR;
```

```
        int answer,answerL,answerR;

        gateL=NOSYNC;gateR=NOSYNC;
        run B1(<S,SYNC_SET>,<gate,gate_type>,rndzvousL,blckL);
        run B2(<S,SYNC_SET>,<gate,gate_type>,rndzvousR,blckR);

Start:
/* Wait for a message only if the child is active */
        if
        ::gateL!=STOPPED -> rndzvousL?(gateL)
        ::gateL==STOPPED -> skip
        fi;

        if
        ::gateR!=STOPPED -> rndzvousR?(gateR)
        ::gateR==STOPPED -> skip
        fi;

/* Synchoronization conditions */
atomic {
        if
        ::gateL==gateR ->
            if
            ::gateL==GATE THAT BELONGS TO HCS -> message=NOSYNC
            ::gateL!=GATE THAT BELONGS TO HCS -> message=gateL
            fi
        ::gateL!=gateR ->
            if
            ::(gateL==STOP || gateL==NOSYNC
```

```
                || gateR==STOP || gateR==NOSYNC) ->
                        message=NOSYNC

            ::!(gateL==STOP || gateL==NOSYNC
                || gateR==STOP || gateR==NOSYNC) ->

/* It is not possible to cycle through this state for ever */
        accept_p1:
                    message=NOT_ESTABLISHED
            fi
        fi
};

        rndzvous!(message);
        blck?(answer);

atomic{
        if
        ::answer==EXIT || answer==RETRY || gateL==gateR ->
                    answerL=answer;answerR=answer
        ::!(answer==EXIT || answer==RETRY || gateL==gateR) ->
                        if
                        ::gateL==STOP || gateL==NOSYNC ->
                                answerL=PROCEED
                        ::!(gateL==STOP || gateL==NOSYNC) ->
                                answerL=RETRY;
                        fi;
                        if
                        ::gateR==STOP || gateR==NOSYNC ->
```

```
                                answerR=PROCEED
                        ::!(gateR==STOP || gateR==NOSYNC) ->
                                answerR=RETRY;
                        fi;
        fi
};

/* Send an answer only if the child is active */
        if
        ::gateL!=STOPPED -> blckL!(answerL)
        ::gateL==STOPPED -> skip
        fi;
        if
        ::gateR!=STOPPED -> blckR!(answerR)
        ::gateR==STOPPED -> skip
        fi;

/* Exit if an EXIT message has been issued or
if both of the children stopped at the same time */
        if
        ::(answer==EXIT || (gateL==gateR && gateL==STOP)) ->
                                    goto Exit
        ::!(answer==EXIT || (gateL==gateR && gateL==STOP)) ->
          atomic {
                if
                ::gateL==STOP -> gateL=STOPPED
                ::gateL!=STOP ->
                        if
                        ::gateR==STOP -> gateR=STOPPED
```

```
                        ::gateR!=STOP -> skip
                        fi
                fi
            }

/* If no child is active isuue a STOP message and exit*/
                if
                ::gateL==gateR && gateL==STOPPED ->
                        rndzvous!(STOP);
                                blck?(answer);
                                goto Exit
                ::!(gateL==gateR && gateL==STOPPED) ->
                                 goto Start
                fi
          fi
Exit:
         skip
}
```

Appendix C

The callD command

C.1 Prologue

This is the code porduced for the **callD**. Initially, both process are instantiated. If in any instanse, we succesfully choose the first action of the right hand process the other one exits. However if the left hand process exits, the other one exits too.

C.2 The code

```
proctype B(<gate,gate_type>;rndzvous;blck)
{
chan rndzvous0[2]=[1] of {int}
chan blck0[2]=[0] of {int}
    int answer,gateL,gateR,answerL,answerR,gate;
    bool INDEX,REFT,RIGHT,RETRY_BOTH=1;

bool DISABLE=1;
    LEFT=0;RIGHT=1;
run B1(<gate,gate_type>,rndzvous0[LEFT],blck0[LEFT])
```

```
run B2(<gate,gate_type>,rndzvous0[RIGHT],blck0[RIGHT])


COLLECT_FIRST_ACTION_OF_RIGHT:
rndzvous0[RIGHT]?(gateR);

START:
rndzvous0[LEFT]?(gateL);
/* Non-deterministic choice between the first
                                action of B2 and B1 */
/* Possible only if the first action B2 is possible */
atomic {
if
::DISABLE && (gateR!=NOT_ESTABLISHED)->
        if
        ::gate=gateL;INDEX=LEFT; goto send_message
        ::gate=gateR;INDEX=RIGHT; goto send_message
        fi
::skip
fi
};
send_message:
rndzvous!(gate);
blck?(answer);

    if
    ::INDEX==RIGHT && answer!=RETRY->
          blck0[LEFT]!(EXIT);
atomic {
```

```
          DISABLE=0;
          LEFT=RIGHT;
          RETRY_BOTH=0
        }
    ::INDEX==RIGHT && answer==RETRY ->
atomic {
        gate=gateL;
        INDEX=LEFT
};
    rndzvous!(gate);
        blck?(answer);
        RETRY_BOTH=1
    ::INDEX!=RIGHT-> RETRY_BOTH=0
    fi;

blckO[INDEX]!(answer);

if
:: answer==EXIT || gate==STOP->
              if
              ::DISABLE -> blckO[RIGHT]!(EXIT)
              ::!DISABLE-> skip
              fi;
              goto Exit
:: answer!=EXIT && gate!=STOP->
               if
               ::RETRY_BOTH==0 -> goto START
               ::RETRY_BOTH==1 ->
                  goto COLLECT_FIRST_ACTION_OF_RIGHT
```

```
                fi
fi;
Exit:
}
```

Appendix D

The callE command

D.1 Prologue

This is the code porduced for **callE**. The algorithm continuously checks if the left hand process has exited. If it has, the left hand process is instantiated.

D.2 The code

```
proctype B(<gate,gate_type>;chan rndzvous;chan blck)
{
chan rndzvous0[1];
chan blck0[0];

int answer,gate;
bool LEFT_CH,READY_RI;


run B1(<gate,gate_type>,rndzvous0,blck0);
LEFT_CH=TRUE;READY_RI=FALSE;
```

```
START:
rndzvous0?(gate)
rndzvous!(gate);
blck?(answer);
blck0!(answer);

/* If the offered action is at gate DELTA prepare to i
nstantiate process B2 as soon as a STOP message has been
issued. After that the left child (LEFT_CH) is
considered to be dead */

if
:: LEFT_CH==TRUE-> if
   ::gate==DELTA && answer==PROCEED ->
READY_RI=TRUE;
   ::gate==STOP && READY_RI==TRUE->
LEFT_CH=FALSE;READY_RI=FALSE;
run B2(<gate,gate_type>,rndzvous0,blck0);
   ::(gate!=DELTA || answer!=PROCEED) &&
(gate!=STOP || READY_RI==FALSE)
-> skip
   fi
:: LEFT_CH==FALSE-> skip
fi;

if
:: answer==EXIT || (!LEFT_CH && gate==STOP)-> goto Exit
:: answer!=EXIT || LEFT_CH || gate!=STOP -> skip
fi;
```

```
goto START;
Exit:
skip
}
```

Bibliography

[AMn88] José A. Mañas. A tutorial on ADT semantics for LOTOS users, part i & ii. Technical report, Dept. Ingeniería Telemática Ciudad Universitaria, 1988.

[AMnDSA93] José A. Mañas, T. Demiguel, J. Salvachua, and A. Azcorra. Tool support to implement LOTOS formal specifications. *Computer networks and ISDN systems*, 25(7):815–839, 1993.

[AMnS92] José A. Mañas and Joaquín Salvachúa. Lambda-beta: A virtual LOTOS machine. *IFIP Transactions C-Communication systems*, 2:441–456, 1992.

[BA89] R. B. Alderlen. Functionality of an ADT interpreter. In P.H J. Van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Decription Technique LOTOS*, pages 397–408. North Holland, 1989.

[BB87] Tommaso Bolognesi and Ed Brinskma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BBD+92] C. Binding, W. Bouma, M. Dauphin, G. Karjoth, and Y. Yang. A common compiler for LOTOS and SDL specifications. *IBM Systems Journal*, 31(4):668–690, 1992.

[CCI89] CCITT. *Functional Specfication and Description Language,(SDL)*, 1989.

[CFGM+92] J. C. Fernandez, H. Garavel, L. Mounier, A. Rasses, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14*. ACM, 1992.

[Cha87] A. Charlseworth. The multi-way rendezvous. *ACM Trans. Programming Languages Syst.*, 9(2):350–366, July 1987.

[CHJ91] N. Hong Cheng, J. P. Hulskamp, and Lindsay N. Jackson. An *occam* implementation of LOTOS communication. In J. Hulskamp, T. Bossomaier, and T. Hintz, editors, *ATOUG-4: The Transputer in Australasia*, pages 27–34, Amsterdam, Netherlands, September 1991. IOS Press.

[CJ78] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, July 1978.

[CVY90] C. Courcoubetis, P. Vardi, M. Wolper, and M Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *2nd Workshop on Computer-Aided Verification*. Rutgers University, June 1990.

[ELS75] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, 1975.

[EM85] H. Ehrig and B. Mahr. Fundamentals of algebraic specification 1. In *EATCS Monographs on Theoretical Computer Science*, volume 6. Springer-Verlag, 1985.

[EW93] H. Eertink and D. Wolz. Symbolic execution of LOTOS specifications. *IFIP transactions C-Communication systems*, 10:295–310, 1993.

[Geo91] C. George. The RAISE specification language - a tutorial. In *Lecture notes in Computer Science*, volume 552, pages 238–319, 1991.

[GG93] Mohamed G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25(9):969–980, 1993.

[Gil88] D. Gilbert. A LOTOS to PARLOG translator. In K. Turner, editor, *Formal Description Techniques*. North Holland, 1988.

[GL93] B. Ghribi and L. Logrippo. A validation environment for LOTOS. *IFIP transactions C-Communication systems*, 16:93–108, 1993.

[GN89] Hubert Garavel and Ellie Najm. TILT: from LOTOS to labelled transition systems. In P.H J. Van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Decription Technique LOTOS*, pages 327–336. North Holland, 1989.

[Gv89] Quiang Gao and Gregor von Bochmann. Distributed implementation of LOTOS multi-rendezvous. In *Protocol Specification, Testing and Verification IX*. North Holland, June 1989.

[Hen88] Mathew Henessy. *Algebraic Theory of Processes*. MIT Press, 1988.

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

[ISO89a] ISO. *IS 8807 Information Proccessing Systems - Open Systems Interconnection: LOTOS- A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1989.

[ISO89b] ISO. *IS 9074 Information Proccessing Systems - Open Systems Interconnection: Estelle- A Formal Description Technique Based on an extended state transition model*, 1989.

[JH91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.

[JT93] Kenneth J. Turner, editor. *Using Formal Description Techniques (An introduction to Estelle, LOTOS and SDL)*. John Willey & Sons, 1993.

[Kal95] Alexandros Kaloxylos. Estopro: An ESTELLE to PROMELA compiler. Master's thesis, Heriot Watt University, 1995.

[Kar92] Gunter Karjoth. Implementing LOTOS specifications by communicationg state machines. In *Lecture Notes in Computer Science 630*, pages 386–400. Springer Verlag, 1992.

[LFH92] L. Logrippo, M. Faci, and M. HajHussein. Introduction to LOTOS. Learning by examples. *Computer Networks and ISDN Systems*, 23(5):325–342, 1992.

[LLdF+94] L. Léonard, G. Leduc, D. de Frutos, L. Llana, C. Miguel, J. Quemada, and Rabay. G. Belgian-Spanish Proposal for a Time Extended LOTOS. Technical report, Université de Liége and Universidad Politécnica de Madrid and Universidad Complutense, Madrid, November 1994.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer and Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. C.A.R. Hoare. Prentice Hall, 1989.

[MS88] N.F. Maxemchuk and K. Sabnani. Probabilistic verification of communication protocols. In *Protocol Specification, Testing and Verification*, volume VII. North Holland, 1988.

[MS92] J. M. Spivey. *The Z Notation*. Prentice-Hall, 1992.

[MV92] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in LOTOS. In *Formal Description Techniques IV*, pages 495–510. North Holland, 1992.

[Peh90] Björn Pehrson. Protocol verification for OSI. *Computer Networks and ISDN systems*, 18(3):185–201, 1990.

[Per94] Douglas Perry. *VHDL*. New York, London, McGraw-Hill, 2nd edition, 1994.

[PPP91] Sarandis Papanagnoy, Kosmas Papachristos, and Nicholaos Petalidis. *The Pygmalion Compiler*. Computer Science Dpt. University of Crete, 1991.

[PS92] J. Parrow and P. Sjödin. Multiway synchronization verified with coupled simulation. In *CONCUR'92*, volume 630 of *LNCS*, pages 518–533. Springer Verlag, 1992.

[QPF90] J. Quemada, S. Pavon, and A. Fernandez. State exploration by transformation with LOLA. In *Lecture notes in computer science*, volume 407, pages 294–302. 1990.

[Tre89] Jan Tretmans. HIPPO: A LOTOS simulator. In P.H J. Van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Decription Technique LOTOS*, pages 391–396. North Holland, 1989.

[Tur93] LOTOS Newsletter, September 1993.

[VASDU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addiwon-Wesley, 1986.

[vE89] Peter van Eijk. The design of a simulator tool. In P.H J. Van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Decription Technique LOTOS*, pages 351–390. North Holland, 1989.

[VSC90] Adriano Valenzano, Riccardo Sisto, and Luigi Ciminiera. Modeling the execution of LOTOS specifications by cooperating extended finite state machines. *IEEE Transactions on Computers*, pages 780–785, 1990.

[VSC91] A. Valenzano, R. Sisto, and L. Ciminiera. A protocol for multirendezvous of LOTOS processes. *IEEE Transactions on Computers*, 40(1):437–446, April 1991.

[WvB90] Cheng Wu and Gregor von. Bochmann. An execution model for LOTOS specifications. In *GLOBECOM 90 - IEEE Global Telecommunications Conference & exhibition*, volume 3, pages 1890–1895. University of Montreal, 1990.