National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

A Model Checker For LOTOS

By

Brahim Ghribi

Thesis Submitted
to the School of Graduate Studies
in partial fulfillment of the requirements
for the Master's degree in Computer Science
under the auspices of the Ottawa-Carleton
Institute for Computer Science

UNIVERSITÉ D' OTTAWA  UNIVERSITY OF OTTAWA

ISBN 0-315-96000-0

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

To my parents

Acknowledgments

I would like to thank all the people who assisted me in completing this work.

First, I would like to express my deepest appreciation to my supervisor, Professor Luigi Logrippo for his support, encouragement, and fruitful discussions throughout my graduate studies.

I would also like to thank Professor Gerald M. Karam from Carleton University and Professor Ali Mili from the University of Ottawa for accepting to be members of my thesis committee.

I also owe special thanks to my colleagues of the University of Ottawa LOTOS group for their useful suggestions, valuable time, and patient listening.

I am most of all very thankful to my parents who gave me an optimistic outlook, a taste of quality, the moral support and encouragement to further my education in Canada.

Finally, I would like to express my gratitude to the Tunisian Government and the Canadian Institute for Telecommunications Research for their financial support.

Abstract

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

CTL (Computation Tree Logic) is a branching-time temporal logic, which can be used to express properties of the system being designed. Efficient algorithms were reported in the literature which make it possible to check whether a given behaviour tree enjoys a property expressed in CTL. Such algorithms constitute what is commonly called "model checking".

The topic of this thesis is the design and implementation of a model checker for LOTOS specifications called LMC (LOTOS Model Checker).

LMC allows users to check whether a specification behaves correctly. To do so, LMC requires a graph model obtained by expanding the LOTOS specification symbolically, and a set of correctness properties describing the requirements behaviour of the system to be checked. These properties are expressed in the branching temporal logic CTL.

We present an introduction to formal description techniques along with a review of some relevant existing work. We then present the technical framework of the branching temporal logic CTL, and discuss some important aspects such as correctness properties of concurrent systems and their classification. We discuss the algorithms used in our model checker together with their application to LOTOS. Finally, we use two examples to illustrate the validation methodology.

Table of Contents

Chapter 1

Introduction

1.1 Background

The domain of communications protocols is growing remarkably due to an extensive amount of work aiming to standardize communications procedures within ISO and CCITT. This work involved large international projects such as the Integrated Services Data Networks (ISDN), the Open System Interconnection (OSI), and standards for telephone switching systems. These systems are very complex, and therefore subject to design errors. Unpredictable or incorrect behaviour may be discovered only at particular circumstances. Furthermore, experience shows that the cost of detecting and correcting errors increases with each step of the development process, and as systems become more complex, this task becomes more difficult.

The use of formal methods throughout a system's development cycle is a very promising direction to pursue in order to minimize design errors. For instance, the desired behaviour of a system can be specified formally at an early stage of the design. Formal methods can also be used to derive an implementation from a formal specification and to test that the implementation provides the specified behaviour. Finally, the formal specification can be considered as an abstract description of the system's behaviour, and therefore be part of the product documentation.

Several Formal Description Techniques (FDTs) such as Estelle [ISO89], LOTOS [ISO8807], and SDL [CCITT87], have been applied to specify unambiguous, precise, and implementation independent communications protocols. There are strong arguments in favor of using FDTs:

(i) They make it possible to formalize unambiguous standard specifications.

(ii) They ease the derivation of designs.

(iii) They support formal analysis and validation methods.

LOTOS (the Language Of Temporal Ordering Specifications) [ISO8807, BB88, LFH92] is an FDT that has been standardized by ISO for distributed systems specifications, especially for protocol and service specifications of OSI. LOTOS combines process algebraic concepts borrowed from CCS [MIL80] and CSP [Hoa85] with ACT-ONE, an algebraic Abstract Data Type formalism [EM85].

Formal specifications written in LOTOS or any of the other FDTs must have well-defined meaning, which requires correct syntax and static semantics. It turns out in practice that many errors are detected even at this early stage and therefore, additional analysis is needed in order to gain more confidence in the formal specification. For this reason, several validation tools have been developed for LOTOS in the past few years to increase its usefulness. The main methods applied for validating specifications can be classified into three categories: interactive simulation, symbolic execution, and formal verification. We will consider each of these categories in the following sections.

1.1.1 Interactive Simulation

Execution of LOTOS specifications is made possible by using suitable interpreters (simulators). The existing interpreters for LOTOS today are usually based on step-by-step execution; for instance ISLA [HH88] and HIPPO [vE89] are used in this way. This means that the specification is executed action by action, and the set of the next possible actions is determined after the execution of an action. The designer plays the role of the environment and resolves non-determinism, by deciding which action should be taken

and by providing the required value expressions. This execution method is time consuming especially if (as usually is the case) there are several alternatives for each action and it becomes tedious if one attempts to trace the execution for several steps.

1.1.2 Symbolic Execution

This mode of execution attempts to derive a symbolic behaviour tree of the specification, without the intervention of the user. Values to be provided by the environment are replaced by symbolic values. In this case, the system attempts to go as far as possible. This means that reduction methods, such as loop detection, have to be applied in order to avoid as much as possible the state explosion problem [GL89,QFP88]. The interesting aspect of this way of operating is that it allows the designer to look at the specification from a different perspective, while preserving its original meaning. The behaviour tree obtained may be in some cases a finite graph or "model" (see below). LOLA (LOTOS Laboratory) [QFP88] and SELA (Symbolic Expander of LOTOS Applications) [Ash92] are two transformation tools for LOTOS that belong to this category.

1.1.3 Formal Verification

The operational semantics of LOTOS and its algebraic properties provide a strong theoretical foundation that supports some formal reasoning about the specification, referred to as *verification* in [LL91]. In the domain of communications protocols, verification is defined in [Wes91] as a procedure that compares a formal specification of communications protocol with the specification of what the protocol is intended to do. A formal proof is performed to verify that the behaviour of the protocol specification corresponds to its required properties. These properties must be expressed formally but not necessarily in the same formalism. There are two main approaches to the verification problem [FGMRS91]: proof-based methods and model-based methods.

Proof-based methods consist of carrying out verification at the specification level using theorem provers. A proof of correctness of the specification using a formal deductive system, consisting of axioms and inference rules is composed.

Model-based methods require the generation of all reachable global states of a system and a requirements specification. Both the global states graph (model) of the system and the requirements specification are compared to check for the absence of design errors such as deadlocks.

At the current state of development, proof-based methods have major drawbacks [GS90]: they are usually very complex and they depend heavily on user's interventions during the steps of the proof. This implies that automation of such techniques is limited to small examples. Furthermore, at the current state of research, it does not appear that the techniques used for sequential programs can be applied to parallel programs efficiently.

Model-based methods have gained more attention since they are more efficient on practical examples and can be fully automated. Their main drawback consists of the state explosion problem, since for complex systems, the number of states is usually infinite [Wes91].

The model-based techniques are subdivided into two classes depending on the nature of the correctness specification to verify [Sif86, FGMRS91]: Behavioural and logical specifications.

Behavioural specifications

The LOTOS specification and the requirements specification are both represented by transition systems. Bisimulation equivalence relations are then used to prove equivalence between them [Mil80]. Examples of tools that follow this approach are Aldebaran [Fer89, Fer90] and Squiggles [BC89]. They allow the comparison between two labelled transition systems using different equivalence relations.

Logical specifications

Desirable properties for the system are expressed in terms of temporal logic formulas. The behaviour of the system is represented by a model consisting of the set of all possible execution sequences. This method is referred to as "model checking" [CES83, CES86]. Temporal logic is used to express correctness specifications while the model

checking technique attempts to prove that the finite state system meets these correctness specifications. The model checking problem is decidable since if needed, we can do an exhaustive search through the paths of the finite state system. For some logics, which have adequate expressive power to capture certain correctness properties, efficient algorithms for model checking can be developed [Emer90]. The complexity of the model checking algorithm was shown to be linear in both the size of the specification and the size of the global graph [CES86]. The main advantage of model checking is that it can be implemented efficiently. However, the behaviour of the system has to be modeled by a state graph which restricts the applicability of the method to finite state systems.

1.2 The University of Ottawa LOTOS "toolkit"

The following tools are the main components of the University of Ottawa LOTOS "toolkit":

(i) ISLA [HH88], an interpreter that helps simulating the specifications and discovering design errors by giving the designer the ability to monitor and trace some execution sequences. ISLA provides a wide range of services [GHL88]. First the syntax and the static semantics of a LOTOS specification are checked and an internal representation is generated. This internal form is used to execute the specification in a step-by-step mode or to generate a symbolic execution tree. The designer has to play the role of the environment and resolves non-determinism, by deciding which next action should be executed and by providing the required value expressions. It is possible to go back to previous points in the execution and explore other alternatives. This allows more flexibility in moving between various states of the execution tree.

(ii) SVELDA (System for Validating and Executing LOTOS Data Abstractions) [Feh87], a tool that allows proofs of equational theorems, interpretation of expressions constructed from some defined data types, and testing the confluence and/or the termination of a conditional term rewriting system that can be produced from a set of conditional equations.

(iii) SELA [Ash92], a tool that generates the behavior tree of a specification (or process) symbolically, that is, without the use of actual values. It takes a specification and generates a symbolic behaviour tree. Also, SELA provides the means to translate the generated tree into a monolithic style specification and other formats useful for in-depth analysis of the original specification.

(iv) LOTEST [Jaou92], an interactive tool that has many functionalities related to testing. The behaviour trees generated by SELA are used as input by LOTEST to generate canonical testers, failure trees and traces, and to check for conformance and trace inclusion between various trees.

1.3 Motivation of The Thesis

The major problem that is encountered when using an interpreter to execute a specification is the large number of feasible paths that can be explored. That is, for a complex system composed of several interacting processes, it is impractical to execute all the possible sequences of actions offered by a specification in a step-by-step mode. On the other hand, little work has been done with regard to LOTOS tools in the area of formal verification. Therefore, we need an efficient way to verify the LOTOS specification with respect to a set of properties expressing correct temporal behaviour. Our primary goal consists of designing and implementing a model checker for full LOTOS (LMC) to verify temporal logic properties on the graph model generated by SELA. The properties are expressed in the branching temporal logic CTL [CE81, CES86].

The existence of our model checker enables a design methodology involving several phases (Figure 1.1):

1. **Initial phase**: the designer starts with the formulation of some informal requirements of the system to be developed.

2. **Specification phase**: the design is then formally specified in terms of a set of interacting processes in a structured top-down way, by using LOTOS.

3. **Generation phase**: this phase deals with the generation of the symbolic behaviour tree from the LOTOS specification. If the tree is not finite, it is cut to some user-specified length.

4. **Verification phase**: at this point, the designer formulates the initial requirements by means of a set of temporal logic properties to be provided to the model checker. The model checker determines if the properties are true or false for the system specified, by exploring the tree generated in the previous phase.

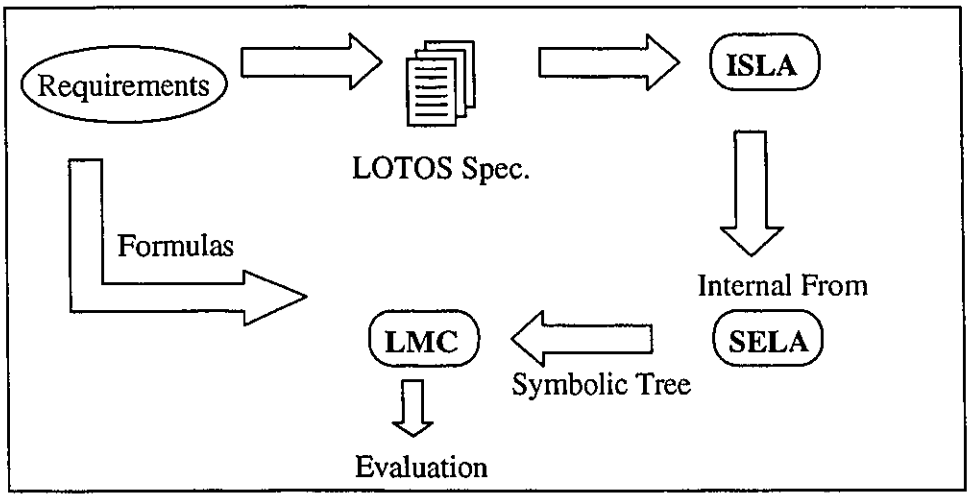


Figure 1.1 The Validation Environment

1.4 The Existing Model Checkers for LOTOS

A few model checkers for LOTOS have been reported in the literature. They are:

(i) The CÆSAR/CLÉOPÂTRE system [GAR89a, GS90, ROD88, RAS90, FGMRS91]. In this system, LOTOS specifications are first compiled into extended Petri Nets using CÆSAR [GAR89a], then into state graphs for verification purposes. CLÉOPÂTRE represents a validation tool for specifications expressed in the branching-time temporal logic LTAC [QS83]. It includes a model checking module [ROD88] and an explanation module [RAS90], which provides diagnostics based on sequences extracted from the graph when a formula is not valid. For reasons that

will be explained later (section 2.4.5), this tool produces extremely large models, although they are generated and stored efficiently.

(ii) The other model checkers that we know of were developed based on the adaptation of existing tools. One of them is built mainly by extending the equivalence verifier Squiggles [BC89] and by using EMC (Extended Model Checker) [CES86] to evaluate CTL formulas [FGLR91]. Squiggles is used to construct a Labelled Transition System from the LOTOS specification. This tool handles only basic LOTOS expressions. It does not check for finiteness and the growth of the LTS can not be controlled. A similar system is reported in [Karj92], but few details are given.

1.5 Organization of the Thesis

This thesis is structured as follows: in chapter 2, we give an overview of the LOTOS language and of the symbolic expansion of LOTOS behaviours. In chapter 3, we present mainly the technical framework of the branching temporal logic CTL, and discuss some important aspects such as correctness properties of concurrent systems and their classification. In chapter 4, we discuss the algorithms used in our model checker. The adaptation of the algorithms to LOTOS is also part of chapter 4. The practical issues related to the design and implementation of LMC are the subject of chapter 5. In chapter 6, we use two examples to illustrate the validation methodology. The conclusions of the thesis along with a discussion of possible future work follow in chapter 7. The LOTOS specifications corresponding to the examples of chapter 6 are provided in appendix A. Appendix B demonstrates the use of LMC to evaluate the properties related to the examples of chapter 6. A diagnostics example is given in appendix C.

Chapter 2

From LOTOS Specifications to Behaviour Trees

We start in this chapter by giving an overview of the specification language LOTOS, then move to discuss the symbolic expansion of LOTOS behaviours using the symbolic expander SELA.

2.1 Introduction

LOTOS (Language Of Temporal Ordering Specification) is a formal description technique standardized 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 [ISO8807].

LOTOS specifications consist of two components: 1) a *control* component based on Milner's Calculus of Communicating Systems (CCS) [Mil80] and Hoare's Communicating 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 data types ACT-ONE [EM85], that describes the data structures and value expressions.

In the next section we give an overview of both components of LOTOS. We concentrate on the control component, and briefly outline the data component as it is outside the scope of this thesis. A more detailed introduction to LOTOS can be found in [BB87, LFH92].

2.2 LOTOS Data Types

The requirement of abstraction from implementation details is one of the main objectives of FDTs. For this reason, LOTOS adopted the Abstract Data Language ACT ONE for defining its data types. Abstract types define only the essential properties and operations of data, without indicating how data values are actually represented and manipulated in memory.

LOTOS is characterized by the following capabilities for specifying abstract data types:

(i) reference to previously defined specifications in a library.

(ii) combinations and extensions of already existing specifications.

(iii) renaming and parametrization of specifications.

(iv) actualization of parametrized specifications.

A data type specification in LOTOS consists mainly of a *signature*, that gives all the information required to build *terms* (also referred to as value expressions), and possibly a list of *equations*.

2.2.1 Signature

The *signature* of a data type specification is the definition of data carriers, referred to as *sorts*, and operations. It includes the domains and ranges of the operations. Consider the following type definition of the natural numbers:

```
type   Naturals
sorts  Nat
opns   0      :          -> Nat
       succ   : Nat      -> Nat
endtype
```

The signature of type *Naturals* includes a unique sort *Nat*, and the operations *0* and *succ*. Operation *0* results in an element of sort *Nat* because it does not have arguments. The operation *succ* can be applied to single elements of sort *Nat*, producing new elements of sort *Nat*. The following terms of sort *Nat* can be constructed:

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

2.2.2 Equations

Equations provide a means to define the semantics of operations. In order to express properties of natural numbers, we need to write some equations . For example, we can use the concept of *equation* to formalize the *plus* operator which denotes the sum of two natural numbers. The extension of type *Naturals* is as follows:

```
type   Naturals
sorts  Nat
opns   0      :              -> Nat
       succ   : Nat          -> Nat
       plus   : Nat, Nat     -> Nat

eqns
       forall x, y: Nat
       ofsort Nat
              plus(x,0)  =  x;
              plus(0, x) =  x;
              plus(x, succ(y)) = succ(plus(x,y));
endtype
```

The first and second equations state that the sum of any natural number x and the natural number *0* is x. The third equation states that the sum of two non-zero natural numbers can be inductively evaluated.

2.3 The Control Component

The control component of LOTOS deals with the description of process behaviours and interactions. The elements of this component are presented in this section.

Distributed concurrent systems are described in LOTOS in a top down hierarchy of process definitions. A typical specification is written as follows:

specification spec_name [g_1,g_2,..,g_n] (v_1,v_2,..,v_m):functionality
behaviour
    < behaviour expression>
where
    <process definitions>
endspec

A process is viewed as a black box interacting with its environment via its observable gates. Its internal actions are unobservable by the environment. The behaviour expression is built by combining LOTOS actions by means of operators and possibly instantiations of other processes. The syntax of a process definition is of the form:

process proc_name [g_1,g_2,..,g_n] (v_1,v_2,..,v_m):functionality
    < behaviour expression>
where
    <process definitions>
endproc

The basic element of a behaviour expression is the action which consists of a gate name associated with a list of experiments, and possibly a predicate that imposes conditions on the event values to be accepted. An event can be the offer of eval(*E*), the value of the *expression E* which is denoted by "*!E*". It can also be of the form "*?x:s*", denoting the readiness to accept a value of sort *s*. For example, if we want to specify that a process accepts a value of sort *Nat* that must be strictly greater than zero at gate *g*, we can write:

g?x:Nat [x > 0]

In general, an action is denoted by:

$$g\ d_1 d_2 ... d_n [P]$$

Where P is a selection predicate, d_i are experiments and g is the gate name. Both d_i and P are optional.

Interprocess communication in LOTOS occurs when two or more processes, having a "rendez-vous" on a gate, agree on one or more values to be established. This is referred to as *matching actions*. Table 2.1 represents a summary of the types of interaction between two processes together with the conditions for, and the effect of interaction. When more than two processes are involved, similar rules apply. Consider two processes A and B, where Process A is prepared to accept any natural number 0, 1, 2, ... at gate g, as denoted in the action $g?x:Nat$, while at the same time process B is ready to accept odd numbers at gate g, which is expressed by the action $g?x:Nat\ [x\ mod\ 2 = 1]$. An interaction will occur at gate g if the environment cooperates by offering odd numbers, namely 1, 3, 5, ...

process 1	process 2	sync. condition	interaction type	effect
$g\ !\ E_1$	$g\ !\ E_2$	eval(E_1) = eval(E_2)	value matching	synchronization
$g\ !\ E$	$g?x:s$	eval(E) $\in$ domain(s)	value passing	after synchronization x = eval(E)
$g?x_1: s_2$	$g?x_2: s_2$	$s_1 = s_2$	value generation	after synchronization $x_1 = x_2 = x$ $x \in$ domain(s_1)

Table 2.1 Types of Interaction

A behaviour expression in LOTOS may contain instantiations of other processes, whose definitions are provided in the "where" clause following the expression.

In the following, we present the basic components of LOTOS behaviour expressions. We recall that a precise definition of the syntax and semantics of LOTOS is given in [ISO8807].

2.3.1 Inaction

In LOTOS, a process can be in a state of deadlock, which means that it cannot offer any action to the environment nor can it perform internal events. The inaction operator *stop* is used to express this fact.

2.3.2 Action prefix

A bahaviour B consisting of a sequence of actions can be written as another behaviour B_2 prefixed by an action using the action prefix operator ";"

$$B = g\ d_1 d_2 ... d_n [P]; B_2$$

The behaviour B may perform independently an internal action that is not observable by the environment, denoted by i, and transform into B_2.

$$B = i; B_2$$

2.3.3 Choice

The choice operator "[]" is used when the environment is able to choose among several actions. A behaviour B can be written in this case as:

$$B = B_1 [] B_2$$

meaning that B can behave as B_1 or B_2. Once the first action of B_1 or B_2 is executed, the choice no longer exists.

Example

B= send_message;(receive_message; exit [] lose_message;stop)

After sending a message, the environment can choose between receiving the message and thus terminating successfully, or losing it in which case it stops. In general, losing the message should not happen unless something "internal" occurs. The behaviour in the example is then modified to:

B= send_message;(receive_message; exit [] i;lose_message;stop)

2.3.4 Parallel composition

A behaviour B can be composed of two behaviours B_1 and B_2 executing independently, except for the actions at any of the gates where B_1 and B_2 must synchronize.

$$B = B_1 \mid [g_1,...,g_m] \mid B_2$$

B_1 and B_2 must synchronize on the actions at gates $g_1,...,g_m$. We can also write:

$$B = B_1 \mid\mid\mid B_2 = B_1 \mid [ ] \mid B_2 \quad \textit{(Interleaving)}$$
$$B = B_1 \mid\mid B_2 = B_1 \mid [g_1,...,g_n] \mid B_2 \quad \textit{(Full synchronization)}$$

where $\{g_1,...,g_n\}$ is the set of all possible gates. In the first case, the synchronization set is empty while in the second case it contains all possible gates of both behaviours.

Example

B_1 = (a;b;c;stop) |[b]| (d;b;e;stop)
B_2 = (a;b;stop) || (a; b;e;stop)
B_3 = (a;b;stop) || (b;a;stop)
B_4 = (a;b;stop) ||| (c;d;stop)

B_1 results in one of the following traces: *a d b c e, d a b c e, a d b e c, d a b e c.*
B_2 results in the unique trace *ab* since the two behaviours cannot synchronize on the action *e*. An immediate deadlock occurs in the case of B_3 and finally, B_4 results in one of the following traces: *a b c d, c d a b, a c b d, a c d b, c a d b, c a b d.*

2.3.5 Hiding

It is possible to hide some actions so that the environment cannot participate in them, by using the "*hide*" operator. The hidden actions become internal to the environment.

$B=$ **hide** $g_1,...,g_n$ **in** B_2

Example

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

results in one of the following execution sequences:

i b c externally observable as *b c*
c i d externally observable as *c d*

2.3.6 Sequential composition (enabling)

The enabling operator ">>" is generally used to express the fact that a behaviour B_1 enables another behaviour B_2 when it terminates successfully.

$B= B_1 >> B_2$

Example

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

This behaviour accepts the sequences *a b d* and *c*.

In general, enabling is associated with passing some parameters that are necessary for the enabled behaviour.

2.3.7 Successful termination

Successful termination of a behaviour expression is denoted by *exit*, which first offers an action on a special internal gate δ, associated with parameters (if any) representing the results, then behaves like *stop*.

$$B = \text{exit}(E_1, \ldots, E_n)$$

$E_1, \ldots, E_n$ are the results of B and will be offered at the special gate δ .

Example

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

This behaviour accepts values for x and y at the gates a and b respectively, offers x and y at gate δ, then stops.

2.3.8 Disabling

The disabling operator "[>" expresses situations where a process can be interrupted by another process during normal functioning.

$$B = B_1 \; [> \; B_2$$

It is possible for B_2 to disable B_1 and start executing unless the latter one has already terminated successfully. Note that B_1 can be interrupted before it even starts to execute, in which case B behaves like B_2.

Example

send_req; receive_req; answer_req; exit [> disconnect

This bahaviour accepts the following sequences:

disconnect
send_req *disconnect*
send_req *receive_req disconnect*
send_req *receive_req answer_req disconnect*
send_req *receive_req answer_req* δ

δ indicates that the left hand side has terminated successfully. Disconnection is not valid any more.

2.3.9 Guarded behaviour

It is possible to impose guards or conditions on a behaviour. The behaviour can be executed only if the guard is evaluated to true.

B= [Guard] -> B_1

B will behave as B_1 if Guard is evaluated to true, and behaves as *stop* otherwise.

Example

```
[ x < 2 ]  -> g!( x + 1);stop
[]
[x > 2] -> exit(x)
```

If x = 1 then the above behaviour is equivalent to *g!2;stop*. If x = 4 then the above behaviour is equivalent to *exit(4)*. If x = 2 then the above behaviour is equivalent to *stop*

2.3.10 Process instantiation

Process instantiation in LOTOS refers to an already defined process, where the associated list of actual gates can be used to rename the formal gate list defining the process. Recursion is possible by making a process refer to itself.

Let $P[g_1,..., g_n](t_1,...,t_m)$ be a process, the following behaviour:

$$B = P[h_1,\ldots, h_n](s_1,\ldots,s_m)$$

behaves as P with the substitution of the formal variable parameters $t_1,\ldots,t_m$ by $s_1,\ldots,s_m$ and with the renaming of the formal gates $g_1,\ldots, g_n$ with the actual gates $h_1,\ldots, h_n$.

Example

Consider the following process declarations:

(i)
```
process in_out [in,out]:=
        in;
        out;
        in_out[out,in]
endproc
```

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

(ii)
```
process out_numbers[give](N:Nat) :=
        [N < 15] -> give!N;
        out_numbers[give](N+1)
endproc
```

The call *out_numbers[give](0)* will generate the following sequence:

give!0 give!1 give!2 ... give!14

□

The reader should note that many syntactic and semantic details of LOTOS were not mentioned above to simplify the overview of the language.

2.4 Symbolic Expansion of LOTOS Specifications

This section deals with the symbolic expansion of LOTOS expressions into behaviour trees[1]. The symbolic expander SELA applies LOTOS inference rules to a behaviour expression. The possible resulting actions and the set of resulting behaviour expressions

[1] We prefer to use the word "tree" despite the fact that the structure may have cycles only for consistency reasons with previous work.

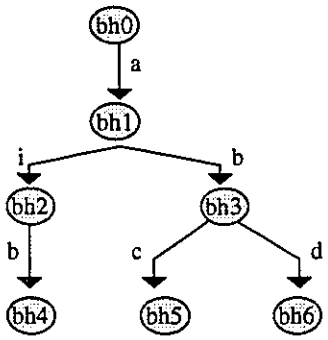
can be obtained. By iteratively applying the same rules on the resulting behaviour expressions, the whole behaviour tree can be generated (if it is finite).

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

For example, given the following LOTOS behaviour:

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

The corresponding behaviour tree is:



bh0 = original behaviour
bh1 = (i ; b ; stop [] b ; (c ; stop [] d ; stop))
bh2 = b ; stop
bh3 = (c ; stop [] d ; stop)
bh4 = bh5 = bh6 = stop

The expansion of a branch ends when one of the following cases is encountered:

(i) Explicit stop or deadlock resulting from synchronization.

(ii) Duplicate behaviours (identical as chains of characters). A loop is detected when a reached behaviour expression has already been generated.

(iii) Upper bounds provided by the user for the width or the depth of the tree.

2.4.1 Explicit Stop or Deadlock

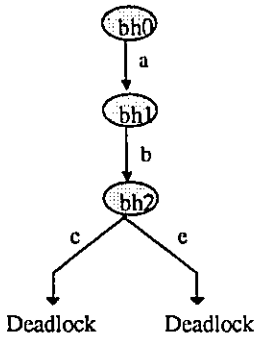
The exploration of a branch will end whenever a stop action in the original behaviour is executed, or a deadlock results from synchronization.

EXAMPLE

Consider the following specification:

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

The corresponding behaviour tree is:



Where the first "Deadlock" is the result of impossible synchronization between actions *d* and *f*, appearing on lines 4 and 8 respectively, while the second is due to the explicit *stops* in the original specification (lines 6 and 10).

2.4.2 Duplicate Behaviours

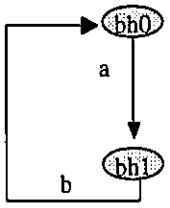
In the most simple case, repetition can be detected whenever a current behaviour is found to be *syntactically equal* to a previous one. Two behaviours are said to be syntactically equal, if they are identical as chains of characters.

EXAMPLE

```
behaviour
        simple1[a,b]
where
        process simple1[a,b]:noexit:=
                a ; b ; simple1[a,b]
endproc
```

The corresponding behaviour tree is:

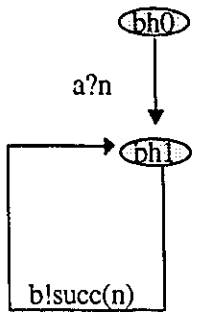
Note that the sequence of actions *a b* is repeated infinitely, and in fact *bh0* and *bh2* are duplicates. Once the duplicate behaviours are detected, the corresponding behaviour tree is:



Consider now the following behaviour:

```
behaviour
        a?n:nat; P1[b](succ(n))
where
process P1[b](n:nat) : noexit :=
        b!n; P1[b](n)
endproc
endspec
```

This example illustrates the case of duplicate behaviours that contain value expressions. The same value *succ(n)* is offered at gate *b* at every recursive call for process *P1*. The behaviour tree can be represented as follows:



2.4.3 Syntactically Equivalent Behaviours

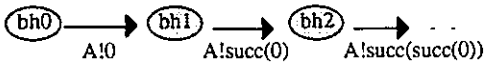
In some cases, behaviours are identical except for the value of some value expressions. This can be due to recursion, where the current behaviour is the instantiation of a process (or a set of processes in the case of parallelism) which had already been instantiated previously. The detection is achieved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

EXAMPLE

Consider the main behaviour:

```
behaviour
        simple2[A](0)
where
process simple2[A](Q:nat):noexit:=
        A!Q;
        simple2[A](succ(Q))
        endproc
endspec
```

In this example, the value offered at gate *A* is incremented at every recursive call of process *simple2* and therefore, the behaviours are identical except for the value offered. The corresponding tree is infinite and can be represented as follows:



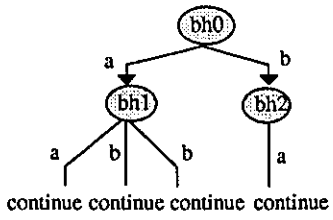
2.4.4 Upper Bounds

Since a behaviour tree can be infinite (this happens when different behaviour expressions keep being generated), the user is required to provide values for the boundaries of the tree, namely, the width (the maximum number of next actions for each node) and the depth (maximum number of successive multiple derivations) of the tree. Therefore, the expansion of a branch stops whenever the width or depth of the tree reaches the maximum values provided by the user. In the case of depth, the node reached is labeled "continue", while in the case of width, the first *N* alternatives (where *N* is the width provided by the user) are generated and the rest are simply truncated.

EXAMPLE

```
specification depth[a,b]:noexit
behaviour
        p[a,b]
where
process  p[a,b]: noexit:=
        a ; p[a,b]
        |||
        b; stop
endproc
endspec
```

If the maximum depth is specified to be 2, the corresponding tree is the following:



2.4.5 Symbolic Values

During the symbolic expansion of LOTOS specifications, some of the derived actions may contain value identifiers yet to be bound by the environment. Each occurrence of such value identifiers is replaced by a *symbolic value* which consists of:

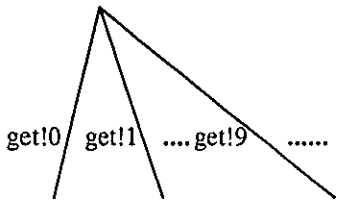
(i) The sort of the value identifier,
(ii) The symbol @ which expresses that the value is to be bound at a gate,
(iii) A numerical value that indicates the binding occurrence of the value.

nat@1, *nat@2*, *BitString@5*, ... are examples of symbolic values. *BitString* and *nat* are the sorts; *nat@1* and *nat@2* can be bound at the same gate but at different times since they are associated with different numerical values.

By using symbolic values, the size of the behaviour tree can be reduced remarkably. This can be explained by the fact that a unique symbolic value can replace an infinite number of actual values. Consider the following simple LOTOS action:

get?x:nat

representing this action symbolically would correspond to a unique edge in the tree labelled with a unique symbolic value (for example *nat@1*). If we choose not to use symbolic values, then the action can be replaced by an infinite set of actions where each action corresponds to a natural number (get!0, get!1, get!2, ...). Graphically, this action corresponds to an infinite number of branches.



Notice that the CÆSAR/CLÉOPÂTRE system produces extremely large models because values are represented explicitly. To solve the problem of infinite values, the user must define the range of each value.

Every variable in the symbolic behaviour tree must be associated with a unique number to differentiate between variables with the same external name and corresponding to different values. To understand the reason for this, we will consider an example.

Example

Consider process *get_money* that keeps accepting coins from the user at gate *coin_in* until he/she pushes a button for a specific pop at gate *buttons*. The button may only be pushed when the user has entered enough money to cover the price of the pop.

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

Suppose we generate the symbolic behaviour tree of the behaviour:

B = *get-money*[*coin-in*, *buttons*](*0*)

If variables are not associated with unique numbers, then it is possible to have the following path in the tree:

coin-in?coin:COIN; coin-in?coin;
buttons?button-name:BUTTON [0+coin+coin >= price(button-name)]

Consider the selection predicate [*0+coin+coin* >= *price*(*button-name*)]. In the expression *coin* + *coin*, no distinction can be made between the first and the second coin, giving the impression that this expression is equivalent to *2*coin*. When numbers are added, the above path instead is shown as:

coin-in?coin@1:COIN; coin-in?coin@2;
buttons?button-name@1:BUTTON[0 + coin@1 + coin@2 >= price(button_name@1)]

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

2.5 Chapter Summary

We presented in this chapter an overview of LOTOS and the symbolic expansion of LOTOS specifications into their corresponding behaviour trees. However, we considered only details that are relevant to our work. For instance, we concentrated on the control component of LOTOS specifications, and briefly outlined the data component as it is outside the scope of this thesis. Also, several useful features of SELA such as the parameterized expansion of LOTOS behaviours were not discussed since they are not of interest to model checking.

Chapter 3

The Computation Tree Logic CTL

We discuss in this chapter the principles of the computation tree logic CTL and its underlying model, kripke structures. We also discuss some important aspects such as correctness properties of concurrent systems and their classification.

3.1 Introduction

Temporal logics were developed for the purpose of describing the ordering of events in time. They were extended from classical logic by including operators that allow the validity of a formula to be dependent on both its normal logical assertion and a time factor. Classical logic can describe a system condition at one particular instant in time while temporal logic can be used to make assertions about the change of system conditions in time during the execution. The underlying nature of time can be viewed in two ways. One way consists of considering time to be deterministic (linear); the other way is to assign to it a branching structure. This leads intuitively to two categories: Linear temporal logic and branching temporal logic.

Each instant has only one possible future or next instant under the assumption of linear time. This yields to a linear sequence of system states. Non-determinism is handled by

assigning a linear state sequence for each possible non-deterministic alternative. In contrast, in a branching temporal logic, time may split into alternate courses representing different possible futures. Nondeterminism is represented naturally as a tree structure of possible state sequences.

Historically, Pnueli was the first to introduce temporal logic as a means of verification of concurrent programs [Pnue77]. His approach consisted of proving manually desired program properties, given a set of program axioms that reflect the behaviour of the program. However, manual construction of such proofs was complex and time consuming. Clarke and Emerson [CE81] gave a polynomial-time algorithm for the branching time logic CTL, and proposed that it could be used as the basis of a practical automatic verification technique they called model checking. Later in [CES86], a more clever version of the algorithm was introduced, and was shown to run in time linear in the length of the input formula and the size of the model. Cavalli and Horn [CH87] gave one of the first examples of using CTL for the proof of temporal logic properties of protocols specified in the formal description technique SDL [CCITT87].

In the following sections, we will introduce some useful definitions and outline the technical framework of the branching time logic CTL. Section 3.2 reflects our work of adapting CTL to our LTS model, while section 3.3 and 3.4 represent previous work.

3.2 Definitions

Labelled transition systems and kripke structures [HC77] are very similar; the difference consists in the labelling on arcs for the former one, and on states for the latter one. The kripke model is very suitable for the verification of properties expressed in a branching temporal logic such as CTL.

3.2.1 Labelled Transition Systems (LTS)

A labeled transition system (LTS) is a 4-tuple $<S,s_0, \Sigma,\Delta>$ where

- S is a finite set of states.
- s_0 is the initial state.

- Σ is a finite set of actions, including the special action *i* called "*internal action*" .
- $\Delta \subseteq S \times \Sigma \times S$ is a labelled transition relation.

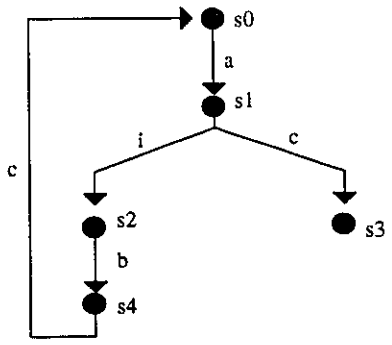
We let s_0, s_1, s_2, ... to range over S and a, b, c, ... to range over Σ.
A labelled transition in Δ is a triple $<s_1, a, s_2>$ usually represented by $s_1 \xrightarrow{a} s_2$.
A state $s \in S$ is called *terminal* if there is no transition from s in Δ.

Example

consider the following LOTOS behaviour:

$$B = a;(c;stop[]i;b;c;B)$$

The corresponding LTS is the following:



- $S = \{s_0, s_1, s_2, s_3, s_4\}$
- $\Sigma = \{a, b, c, i\}$
- $\Delta = \{ <s_0, a, s_1>, <s_1, i, s_2>, <s_1, c, s_3>, <s_2, b, s_4>, <s_4, c, s_0> \}$

3.2.2 Kripke Structures (KS)

A kripke structure is a labelled state transition graph. The definition may vary in the literature. We adapt the most common definition.

A kripke structure is a 5-tuple <K, q_0, Prop, L, $\Re$> where

- K is a finite set of states.
- q_0 is the initial state.
- Prop is a finite, non empty set of atomic propositions.
- L is an assignment of atomic propositions to states ($L : K \rightarrow 2^{Prop}$).
- $\Re \subseteq K \times K$ is a total transition relation.

We let q_1, q_2, q_3, ... range over K and a, b, c, ... over Prop.
A transition is a couple $<q_1, q_2>$ usually represented by $q_1 \rightarrow q_2$.

Algorithms to translate labelled transition systems into kripke structures have been provided [JKP89, DV90]. The next section describes the algorithm introduced in [JKP89].

3.2.3 Transformation from LTS to KS

The semantics of CTL formulas are defined with respect to kripke structures (section 3.3.2). For this reason, it is necessary to transform the labelled transition system associated with the LOTOS specification into the corresponding kripke structure on which desired properties can be proven. Actions that are arcs in the LTS become nodes in the kripke structure. The transformation algorithm described below preserves strong bisimulation equivalence [Mil80] between labelled transition systems and kripke structures; that is, strong bisimulation equivalent labelled transition systems are mapped into strong bisimulation equivalent kripke structures [FGLR91].

Let $\Im = <S, s_0, \Sigma, \Delta>$ be a labelled transition system. We obtain the kripke structure $ks = <K, q_0, Prop, L, \Re>$ as follows:

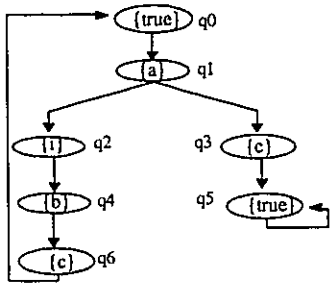
- The set Prop of atomic propositions is defined as Σ (Prop = Σ).
- For every pair of transitions $s_1 \xrightarrow{a} s_2$, $s_2 \xrightarrow{b} s_3$ in the LTS, $<s_1 \xrightarrow{a} s_2, s_2 \xrightarrow{b} s_3>$ is in $\Re$.
  For every transition $s_0 \xrightarrow{a} s$ in the LTS, the pair $<q_0, s_0 \xrightarrow{a} s>$ is in $\Re$.
  For every transition $s_1 \xrightarrow{a} s_2$ for which no transition is possible from s_2 in the LTS, the pair $<s_1 \xrightarrow{a} s_2, q_f>$ is in $\Re$.
  If q_f is in K, then $<q_f, q_f>$ is in $\Re$.

$$\begin{aligned}\Re = \quad & \{<s_1 \xrightarrow{a} s_2,\ s_2 \xrightarrow{b} s_3> \mid s_1 \xrightarrow{a} s_2,\ s_2 \xrightarrow{b} s_3 \in \Delta\} \\ & \cup \{<q_0,\ s_0 \xrightarrow{a} s> \mid s_0 \xrightarrow{a} s \in \Delta\} \\ & \cup \{<s_1 \xrightarrow{a} s_2,\ q_f> \mid s_1 \xrightarrow{a} s_2 \in \Delta \text{ and } q_f \text{ is terminal in ks}\} \\ & \cup \{<q_f,\ q_f>\}.\end{aligned}$$

- Let $L(s_1 \xrightarrow{a} s_2) = \{a\}$ for $s_1 \xrightarrow{a} s_2 \in \Delta$, and
  $L(q_0) = L(q_f) = \{\text{true}\}$.

□

Consider the same behaviour defined in the example above. By applying the transformation algorithm, we get the following kripke structure:



- $K = \{ q_0, q_1, q_2, q_3, q_4, q_5, q_6 \}$
- $L = \{\{\ \}, \{\ a\ \}, \{c\}, \{i\}\}$.
- $\Re = \{ <q_0, q_1>, <q_1, q_2>, <q_1, q_3>, <q_3, q_5>, <q_2, q_4>, <q_4, q_6>, <q_6, q_1>, <q_5, q_5>\}$.

State q_5 is a *terminal* state, it corresponds to *stop* in the LOTOS behaviour of the example. We let $<q_5, q_5> \in \Re$ since the latter one must be total, i.e., $(\forall\ q \in K)\ (\exists\ q' \in K)\ [<q, q'> \in \Re]$.

In the next section, we introduce the definition of CTL, then derive the satisfaction relation for each of its operators.

3.3 Computation Tree Logic (CTL)

CTL is closely related to branching-time logics proposed in [Lamp80, EC81, BAPM83] and was itself proposed in [CE81].

3.3.1 Definition (CTL)

Let Prop denote a set of atomic propositions, the set of CTL formulas is defined recursively as follows:

- Every atomic proposition p ∈ Prop is a CTL formula.
- if f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, AX(f_1), EX(f_1), A[f_1 υ f_2], E[f_1 υ f_2].

AX(f_1) means that f_1 holds in every immediate successor of the current state.
EX(f_1) means that f_1 holds in some immediate successor of the current state.
A[f_1 υ f_2] means for every computation path, starting at the current state, there exists a sequence of transitions satisfying f_2 at last, and f_1 for all the other transitions.
E[f_1 υ f_2] means for some computation path, starting at the current state, there exists a sequence of transitions satisfying f_2 at last, and f_1 for all the other transitions.

3.3.2 Satisfaction Relation of CTL

The semantics of CTL formulas are in fact defined with respect to a kripke structure <K, q_0, Prop, L, $\Re$>. A *Computation path* is an infinite sequence of states (q_0, q_1, q_2,...) such that ($\forall$ q_i ∈ K) ($\exists$ q_{i+1} ∈ K) [<q, q_{i+1}> ∈ $\Re$].

Let ks = <K, q_0, Prop, L, $\Re$> be a kripke structure, $q_0 \models_{ks} f$ means that formula f holds at state q_0 in ks. We will simply write $q_0 \models f$. The satisfaction relation is defined inductively as follows:

$q_0 \models p$	iff $p \in L(q_0)$.
$q_0 \models \neg f$	iff not($q_0 \models f$).
$q_0 \models f_1 \wedge f_2$	iff $q_0 \models f_1$ and $q_0 \models f_2$.
$q_0 \models$ AX(f)	iff $\forall$ q such that $< q_0, q> \in \Re$, $q \models f$.
$q_0 \models$ EX(f)	iff $\exists$ q such that $< q_0, q> \in \Re$, $q \models f$.
$q_0 \models$ A[f_1 υ f_2]	iff for all paths (q_0, q_1, q_2,...) from q_0 in ks $\exists$ i such that 1. $q_i \models f_2$, and 2. $\forall$ j such that $0 \leq j < i$, $q_j \models f_1$.

$q_0 \models E[f_1 \upsilon f_2]$ iff for some path $(q_0, q_1, q_2, \ldots)$ from q_0 in ks
$\exists$ i such that
1. $q_i \models f_2$, and
2. $\forall$ j such that $0 \leq j < i$, $q_j \models f_1$.

□

The following abbreviations are also used in writing CTL formulas:

$AG(f) \equiv \neg AF(f)$ means that f holds at every state for every future path; that is *f holds globally.*

$EG(f) \equiv \neg EF(\neg f)$ means that f holds at every state for some future path.

$AF(f) \equiv A[True \upsilon f]$ means that f holds in the future along every path from q_0; in other words f is *inevitable.*

$EF(f) \equiv E[True \upsilon f]$ means that f holds along some future path from q_0; in other words *f potentially holds.*

Figure 2.1 shows how these operators would be represented in a tree structure.
The "next time" and "until" operators are taken as primitives, and the other operators are defined as abbreviations. However, they are of sufficient importance that their formal definitions can be given explicitly as follows:

$q_0 \models AG(f)$ iff for all paths $(q_0, q_1, q_2, \ldots)$ from q_0 in ks
$\forall i \geq 0 \quad q_i \models f$.

$q_0 \models EG(f)$ iff for some path $(q_0, q_1, q_2, \ldots)$ from q_0 in ks
$\forall i \geq 0 \quad q_i \models f$.

$q_0 \models AF(f)$ iff for all paths $(q_0, q_1, q_2, \ldots)$ from q_0 in ks
$\exists i \geq 0 \quad q_i \models f$.

$q_0 \models EF(f)$ iff for some path $(q_0, q_1, q_2, \ldots)$ from q_0 in ks
$\exists i \geq 0 \quad q_i \models f$.

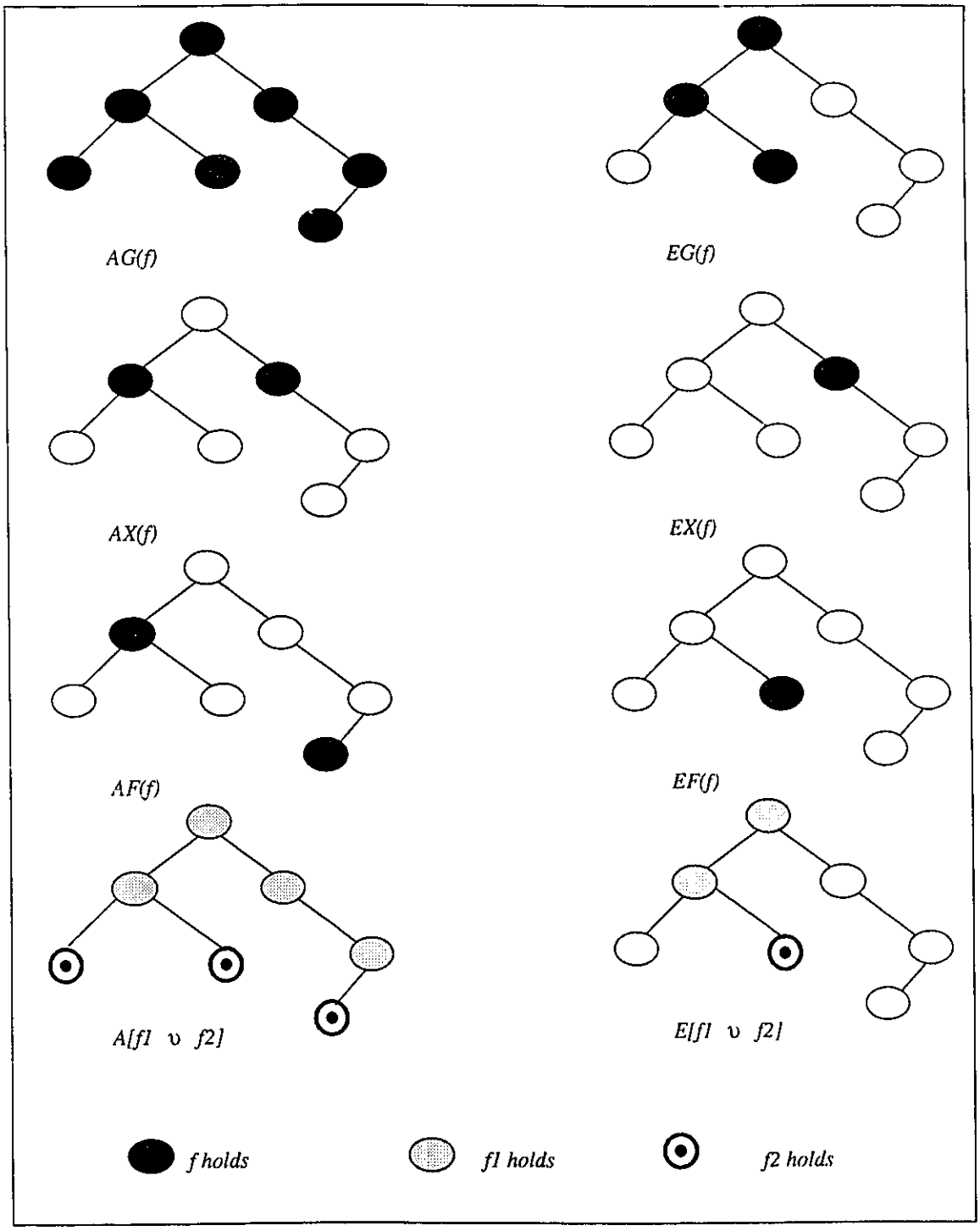


Figure 2.1 CTL Operators

3.3.3 Validity and Satisfiability of Formulas

Under the framework of temporal logic, the notions of *validity* and *satisfiability* of a formula are often encountered. Precise definitions of both terms were given in [Emer90].

A formula f is said to be *satisfiable* if there exists a structure M that is a model of f (i.e. there is some state s in M such that: $s \models_M f$).

A formula f is *valid* if every structure M is a model of f (i.e. for every structure M and every state s in M we have $s \models_M f$).

We have the following examples to illustrate the difference between validity and satisfiability of a formula:

(i) $f_1 \Rightarrow \mathrm{EX}(f_2)$ intuitively means that "if f_1 is true now then f_2 will be true at some immediate next moment". In general, this formula is satisfiable but not valid.

(ii) $f_1 \wedge \mathrm{AG}(f_1 \Rightarrow \mathrm{AX}(f_1)) \Rightarrow \mathrm{AG}(f_1)$ means that "if f_1 is true now and whenever f_1 is true it is also true at every next moment, then f_1 is always true". This formula is valid.

(iii) $\mathrm{A}[f_1 \cup f_2] \wedge \neg(f_1 \wedge f_2)$ means that "f_1 will be always true until f_2 eventually holds, and f_1 and f_2 are false now ". This formula is not satisfiable.

(iv) $\mathrm{EF}(f_1 \wedge f_2)$ means "f_1 and f_2 will be true at some point in the future". This formula is obviously satisfiable but not valid.

□

Proving the validity of a formula is outside the scope of this thesis since such task requires theorem proving. Our goal is to check whether a certain formula is satisfied on a given model of a LOTOS specification.

CTL can be used as a specification language to specify correctness properties of a system under design. In the following section, we discuss these correctness properties and show how CTL can be used to express such properties.

3.4 Correctness Properties

The definition of system correctness usually depends on the type of system to be designed. In the context of communication protocols, a system is *correct* if it satisfies the following conditions:

(i) It is free of logical errors. This means that the various functions provided by the system have to be consistent with the intended requirements.

(ii) Its behaviour is free of temporal errors. These include deadlock, violation of mutual exclusion, etc.

For sequential programs, the main interest is usually the result. Verification attempts to prove that a sequential program terminates and outputs the right result. In the case of concurrent systems, termination is not the major issue since they are in general meant to execute continuously. Examples of correctness properties in communications protocols would be for example whether a request for a message is provided or whether messages exchanged between processes are not erroneous. The set of these correctness properties is also referred to as *partial specification* [CP88] since only limited aspects of system behaviour are specified. However, if the properties are carefully chosen to cover fundamental aspects of the system's behaviour, then we can be confident that the system will do something correctly.

Correctness properties usually fall into two categories [Pnue77, OL82]: "safety" properties also known as "invariance" properties, and "liveness" properties also referred to as "eventuality" or "progress" properties. We will consider the descriptions of safety and liveness separately below, and use CTL to express some correctness properties.

Note that these categories are not generally considered exclusive and authors disagree on what properties belong to each of the categories. We follow the classification given in [Emer90, MP81b].

3.4.1 Safety Properties

This class of properties expresses that "nothing bad will happen". This means that the system will never engage in an undesired state. These properties hold continuously throughout execution and for this reason, they are also referred to as "invariance" properties. Examples of safety properties include:

Absence of deadlock

A deadlock occurs in a concurrent system if no process can execute a next action. This means that the system cannot progress from the current state. This case is referred to as an *absolute deadlock* in [Pnue81]. A partial subset of the system processes may be caught in a *local deadlock* while the rest of the processes are functioning normally. The following formula captures freedom from deadlock for a concurrent system with n processes:

$$AG(enabled_1 \vee enabled_2 \vee ... \vee enabled_n)$$

where $enable_i$ means that process i is enabled.

Mutual exclusion

This property assures that processes will not be able to access a common resource at the same time. For example, the requirement for mutual exclusion to the *critical-section* problem can be written:

$$AG(\neg(CS_1 \wedge CS_2))$$

where CS_i indicates that control of process i is at its critical section.

Global invariance

The invariance of some properties may be independent of any particular execution state of the system. In other words, they must hold no matter what the system does [MP81b].

For example, for a bounded channel, a global invariance property could assert that the maximum capacity of the channel should not be exceeded.

3.4.2 Liveness Properties

These properties assure that "something good will eventually happen". This means that the system will be able to reach desirable states. Examples of Liveness properties include:

Guaranteed accessibility

If a process has a critical section then this property asserts that if the process wishes to enter its critical section, it will eventually be allowed access [MP81b]. Properties of this category can be expressed generally as follows:

$$AG(Try_i \Rightarrow AF(CS_i))$$

Where Try_i and CS_i indicate that process i is trying and is in its critical section respectively.

Responsiveness

Interactions between system processes often involve requests for services and exchange of information. Responsiveness properties assure that requests for services are granted. A property ensuring that requests for services are granted is captured by an assertion of the form:

$$AG(Req_i \Rightarrow AF(Grant_i))$$

where Req_i and $Grant_i$ are predicates indicating that a request by a process is made and a grant for access to process i is provided respectively.

Absence of starvation

Starvation is defined as a situation in which some process(es) cannot proceed even though the system may still progress by having other processes execute [MP81b]. In general, this property can be expressed by asserting that a process will not remain in the same state.

Another general type of correctness properties is referred to as *precedence properties* [Emer90]. These properties deal with temporal ordering, precedence, or priority of events. For example, if we want a $Grant_i$ to be issued only if preceded by a Req_i, we can write:

$$AG(\neg Req_i \Rightarrow A[\neg Grant_i \upsilon Req_i]).$$

We can write $(p\ B\ q)$ to denote "p before q". The *before* operator can be derived using the *until* operator as follows:

$$p\ B\ q \equiv \neg((\neg p) \upsilon q)$$

which means "it is false that p does not occur until q becomes true". Therefore, we can generalize by using the universal quantifiers as follows:

$$A[p\ B\ q] \equiv \neg E[(\neg p) \upsilon q]$$

If we want to express that every grant for a process i is to be issued only if preceded by a request, we can write:

$$A[Req_i\ B\ Grant_i]$$

To illustrate the property of First-In First-Out (FIFO) responsiveness in a queue, we can write:

$$(Req_i\ B\ Req_j) \Rightarrow (Grant_i\ B\ Grant_j)$$

□

3.5 Chapter Summary

We presented in this chapter mainly the technical framework of the branching temporal logic CTL. We started by giving some useful definitions, then outlined the algorithm to transform labelled transition systems into kripke structures. Both the definition of CTL and the satisfaction relation of each of its operators were outlined. Also, we discussed two important notions encountered often in the framework of temporal logic: validity and satisfiability of formulas. We have used some examples to illustrate the difference between them. Finally, we have discussed correctness properties and their classification and we have used CTL to express some examples of these properties.

Chapter 4

The Model Checking Algorithms

We discuss in this chapter the model checking algorithms used in the development of our model checker. The adaptation of the algorithms to LOTOS is also discussed and a modified satisfaction relation for CTL is given to deal with finite and incomplete computations.

4.1 The Checking Algorithms

The checking algorithms for the temporal operators are based on the *fixpoint characterization* of their definitions. Each of the operators can be defined in terms of a condition in the current state and a condition along next time states. The fixpoint characterization for the CTL operators is given below:

$$
\begin{aligned}
\mathrm{AG}(f) &\equiv f \wedge \mathrm{AX}\,(\mathrm{AG}(f)) \\
\mathrm{EG}(f) &\equiv f \wedge \mathrm{EX}\,(\mathrm{EG}(f)) \\
\mathrm{AF}(f) &\equiv f \vee \mathrm{AX}\,(\mathrm{AF}(f)) \\
\mathrm{EF}(f) &\equiv f \vee \mathrm{EX}\,(\mathrm{EF}(f)) \\
\mathrm{A}[f_1 \upsilon f_2] &\equiv f_2 \vee (f_1 \wedge \mathrm{AX}\,(\mathrm{A}[f_1 \upsilon f_2])) \\
\mathrm{E}[f_1 \upsilon f_2] &\equiv f_2 \vee (f_1 \wedge \mathrm{EX}\,(\mathrm{E}[f_1 \upsilon f_2]))
\end{aligned}
$$

□

The next time operators do not have fixpoint characterizations since their validity depends only on the next states.

The model checking algorithms are based on a depth-first search starting at the current state s_0 and proceeding through the branches of the tree. The search is made according to the definitions of the temporal operators and terminates when the satisfiability of the formula has been established. The algorithms are explained in pseudo code format. The boolean function *holds(f,s_i)* is set to true when the formula f is true at state s_i. The boolean flag *visited(s_i)* is set to true when state s_i is visited during the depth-first search. At the beginning, *visited(s_i)* is set to false for every state in the model.

4.2.1 The Next Time Operators

The truth value of *AX(f)* at state s_i depends on the truth value of f at every immediate next state. The boolean function *holds* is initialized to *true* (line 1) and a loop is applied to check each immediate successor (lines 2 to 7). The loop terminates if f is false at an immediate successor state (line 3) or all the immediate successors were checked. The second case means that *AX(f)* holds at state s_i otherwise the loop would have halted already.

```
1. holds(AX(f),s_i) ← true
2. for all successors t of s_i such that (s_i,t) ∈ ℜ do
3.         if ¬holds(f,t) then
4.                 holds(AX(f),s_i) ← false
5.                 return
6.         endif
7. endfor
8. return
```

□

The algorithm for *EX(f)* is described in the same fashion. However, the boolean function *holds* is set to *false* first, and the loop stops when f is found to be true at some immediate successor state or no successor state is found to satisfy this condition. The second case implies that *EX(f)* is not satisfied.

```
1.  holds(EX(f),s_i) ← false
2.  for all successors t of s_i such that (s_i,t) ∈ ℜ do
3.          if holds(f,t) then
4.                  holds(EX(f),s_i) ← true
5.                  return
6.          endif
7.  endfor
8.  return
```

□

4.2.2 The Global Operators

The global operators *AG(f)* and *EG(f)* express the occurrence of events globally. *AG(f)* states that *f* holds at every state of the model whereas *EG(f)* states that there is some path, at which *f* holds at each state. At the beginning, *visited(s_i)* is set to false for all the states. The algorithm for *AG(f)* can be written as follows:

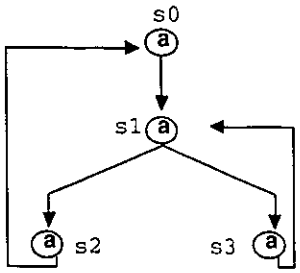
```
 1.  if visited(s_i) then
 2.          holds(AG(f),s_i) ← true
 3.          return
 4.  else
 5.          visited(s_i) ← true
 6.          if holds(f,s_i) then
 7.                  for all successors t of s_i such that (s_i,t) ∈ ℜ do
 8.                          if ¬ holds(AG(f),t) then
 9.                                  holds(AG(f),s_i) ← false
10.                                  return
11.                         endif
12.                 endfor
13.                 holds(AG(f),s_i) ← true
14.                 return
15.         else
16.                 holds(AG(f),s_i) ← false
17.                 return
18.         endif
19. endif
```

□

Note that the algorithm corresponds exactly to the fixpoint characterization given above: *AG(f)* holds if *f* holds in the current state and *AG(f)* holds for all next states.

We recall that the semantics of CTL are defined with respect to a kripke structure (section 3.3.2). The kripke structure has to be finite, and each state must have at least one successor due to the fact that the transition relation is total. This implies that the depth-first search of a branch ends eventually with a state that has been visited previously. This fact is illustrated in the first part of the algorithm (lines 1 to 3). Having reached a state that has already been visited means that no states have been found where f is false otherwise the algorithm would have halted. The algorithm can conclude that $AG(f)$ is true for the current branch under search. The recursive call to the boolean function *holds* (line 8) causes a depth-first search for each of the successors of the current state under checking. If at any state, the immediate assertion f is found to be false (line 15), then the function is set to false and the value is propagated back to the initial part of the algorithm causing it to halt.

To illustrate how the algorithm works, we will check the formula $AG(a)$ on the following model:



The algorithm starts at state s_0. State s_0 has not been visited yet, therefore *visited*(s_0) is set to true and since a is *true* at s_0, the boolean function *holds* is called recursively for each immediate successor (only state s_1 in this case). In the same way, state s_1 is marked as visited and a is found *true* at s_1 therefore, *holds* is called for states s_2 and s_3. Eventually, state s_0 will be reached coming from state s_2, and since it has been visited already, a return to the last call (state s_2) is made. No other successors of s_2 are found therefore, the return is propagated back to the previous call for *holds* (state s_1). State s_3 is not visited yet, so the a recursive call for *holds* is made, etc. The algorithm continues until a is found true at all the states or it is false at a certain state during the depth-first search. In our example $AG(a)$ is satisfied since a is true at all the states.

After initializing *visited(s_i)* to false for every state, the algorithm for *EG(f)* can be written in the same fashion as follows:

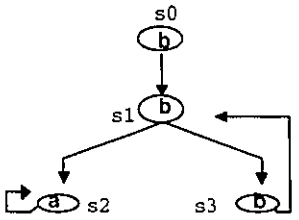
```
1.  if visited(s_i) then
2.          holds(EG(f),s_i) ← true
3.          return
4.  else
5.          visited(s_i) ← true
6.          if holds(f,s_i) then
7.                  for all successors t of s_i such that (s_i,t) ∈ ℜ do
8.                          if holds(EG(f),t) then
9.                                  holds(EG(f),s_i) ← true
10.                                 return
11.                         endif
12.                 endfor
13.                 holds(EG(f),s_i) ← false
14.                 return
15.         else
16.                 holds(EG(f),s_i) ← false
17.                 return
18.         endif
19. endif
```

□

The checking algorithms for *AG(f)* and *EG(f)* are similar, the main differences being the conditions for which the algorithm returns false. For *AG(f)*, the algorithm returns false at a state if the immediate assertion of *f* is found false at that state, or recursively if *AG(f)* is false at some successor of that state. For *EG(f)*, the algorithm returns false at a state if the immediate assertion of *f* is found false at that state, or recursively if *EG(f)* does not hold for every successor of that state (otherwise a successor would have been found for which *EG(f)* holds).

Example

The following tree is a model of *EG(b)* but not of *AG(b)*:



□

4.2.3 The Future Operators

The future operators are used to express the occurrence of events in the future without giving information about when exactly these events are to happen. The operator *AF(f)* states that *f* is inevitable (i.e. *f* will be true for every path in the model, at some point in the future). The operator *EF(f)* states that *f* will be true at some path in the future. The algorithm for *AF(f)* can be written as follows:

```
1.  if visited(s_i) then
2.          holds(AF(f),s_i) ← false
3.          return
4.  else
5.          visited(s_i) ← true
6.          if holds(f,s_i) then
7.                  holds(AF(f),s_i) ← true
8.                  return
9.          else
10.                 for all successors t of s_i such that (s_i,t) ∈ ℜ do
11.                         if ¬holds(AF(f),t) then
12.                                 holds(AF(f),s_i) ← false
13.                                 return
14.                         endif
15.                 endfor
16.                 holds(AF(f),s_i) ← true
17.                 return
18.         endif
19. endif
```

We have seen that in the global operators algorithms, when a state is revisited, it means that a cycle was detected where the formula holds in every state, and the algorithm returns *true* for the revisited state. In the case of the future operators, a state that has been visited indicates that a cycle has been detected where f does not hold. This is expressed in the first part of the algorithm (lines 1 to 3). The algorithm ends successfully for *AF(f)* when a state is reached for each possible path, where f is true. If f is false at a state, a recursive call to check every successor is made (line 10).

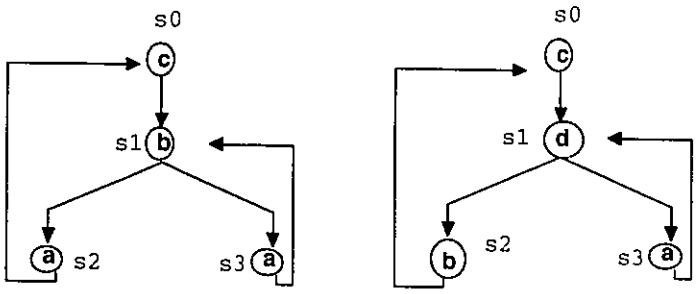
Similarily, we can derive the algorithm for *EF(f)* as follows:

```
 1.  if visited(s_i) then
 2.          holds(EF(f),s_i) ← false
 3.          return
 4.  else
 5.          visited(s_i) ← true
 6.          if holds(f,s_i) then
 7.                  holds(EF(f),s_i) ← true
 8.                  return
 9.          else
10.                  for all successors t of s_i such that (s_i,t) ∈ ℜ do
11.                          if  holds(EF(f),t) then
12.                                  holds(EF(f),s_i) ← true
13.                                  return
14.                          endif
15.                  endfor
16.                  holds(EF(f),s_i) ← false
17.                  return
18.          endif
19.  endif
```

□

Example

Tree (1) is a model of *AF(a)* and *AF(b)*. Tree (2) is a model of *EF(a)*, *EF(b)*, *AF(c)*, and *AF(d)* but not of *AF(a)* nor *AF(b)*:



□

4.2.4 The Until Operators

The until operators are used to express the occurrence of some events before others. The operator $A[f_1 \upsilon f_2]$ states that f_1 holds at every state, in every path, until f_2 becomes true. The operator $E[f_1 \upsilon f_2]$ means that there exists some path where f_1 holds at every state, in every path, until f_2 becomes true. The algorithm for $A[f_1 \upsilon f_2]$ is the following:

```
1.  if visited(s_i) then
2.          holds(A[f_1 υ f_2],s_i) ← false
3.          return
4.  else
5.          visited(s_i) ← true
6.          if  holds(f_2,s_i)  then
7.                  holds(A[f_1 υ f_2],s_i) ← true
8.                  return
9.          else
10.                 if ¬ holds(f_1,s_i) then
11.                         holds(A[f_1 υ f_2],s_i) ← false
12.                         return
13.                 endif
14.         endif
15.         for all  successors t of s_i such that (s_i,t) ∈ ℜ do
16.                 if ¬ holds(A[f_1 υ f_2],t) then
17.                         holds(A[f_1 υ f_2],s_i) ← false
18.                         return
19.                 endif
20.         endfor
21.         holds(A[f_1 υ f_2],s_i) ← true
22.         return
23. endif
```

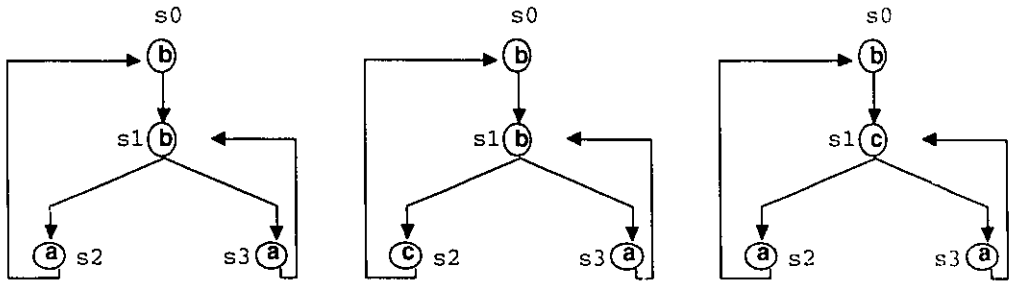
For $A[f_1 \upsilon f_2]$, the algorithm returns false when a visited state is reached again. This implies that a cycle has been detected on which f_1 holds but f_2 is never reached (lines 1 to 3). It should be noticed that if f_2 holds at a particular state, then $A[f_1 \upsilon f_2]$ is *true* at that state and therefore, we don't need to proceed any further from that state (lines 6 to 8). The algorithm returns false also when a state is reached where f_1 is false (lines 10 to 13). When f_1 is true at a state but not f_2, all the successors are checked recursively (lines 15 to 18).

The algorithm for $E[f_1 \upsilon f_2]$ is very similar, except for the case when f_1 is found true at state s_i. In this case, it is sufficient to find some successor for which $E[f_1 \upsilon f_2]$ recursively holds. The algorithm can be written as follows:

```
1.  if visited(s_i) then
2.          holds(E[f_1 υ f_2],s_i) ← false
3.          return
4.  else
5.          visited(s_i) ← true
6.          if  holds(f_2,s_i)  then
7.                  holds(E[f_1 υ f_2],s_i) ← true
8.                  return
9.          else
10.                 if  ¬ holds(f_1,s_i) then
11.                         holds(E[f_1 υ f_2],s_i) ← false
12.                         return
13.                 endif
14.         endif
15.         for all  successors t of s_i such that (s_i,t) ∈ ℜ do
16.                 if  holds(E[f_1 υ f_2],t) then
17.                         holds(E[f_1 υ f_2],s_i) ← true
18.                         return
19.                 endif
20.         endfor
21.         holds(E[f_1 υ f_2],s_i) ← false
22.         return
23. endif
```

□

Example



Tree (1) is a model of $A[b \upsilon a]$. Tree (2) is a model of $E[b \upsilon a]$ but not of $A[b \upsilon a]$. Tree (3) is a model of $A[b \upsilon c]$ but not of $E[b \upsilon a]$.

□

4.2 Complexity of The Algorithms

Given a kripke structure <K, q_0, Prop, L, $\Re$>, The model checking algorithms depend on the number of the states of the structure (*Card(K)*), the number of transitions between the states (*Card($\Re$)*) and the length of the formula *F*. The actual complexity and the proof of correctness of the algorithms were given in [CES86]. The complexity of the algorithms is:

$$O(length(F) \times (Card(K) + Card(\Re))$$

The length of a formula *F* is equal to the number of simple CTL terms and logical connectives ($\Rightarrow$, $\vee$, $\wedge$) of which it is composed. For example, *length(AG(a))* is equal to 1, and *length(AG(b $\Rightarrow$ EF(c $\vee$ a)))* is equal to 5.

We notice that checking the satisfiability of *AG(a)* for example, requires the recursive call to the function *holds* at least *(Card(S) + Card($\Re$))* times. That is, to determine the satisfiability of *AG(a)* at s_i, we need to determine the immediate assertion of *a* at s_i and at each successor of s_i etc. Determining the immediate assertion of *a* explains why we need *Card(S)*, and checking every successor of each state is the reason behind adding *Card($\Re$)*. Each state s_i will be visited at most twice. The first time to determine whether

$AG(a)$ is true at the root state s_0, at which time $visited(s_i)$ is set to true. The second time to determine the satisfiability of $AG(a)$ at s_i. The second time represents the case of detection of a cycle. If the state s_i has previously been visited, then we conclude that $AG(a)$ is true at s_i (otherwise the algorithm would have halted already if $AG(a)$ was false at a previously visited state). Hence, checking the complete structure for $AG(a)$ requires at most:

$$2 \times (Card(K) + Card(\Re))$$

A formula of arbitrary length would require calling the appropriate checking algorithm once for each simple CTL term of which the formula is composed plus once more for each logical connective used. This explains why we need to multiply by the length of the formula.

The complexity of the other operators can be determined in the same fashion and is of comparable order.

4.3 Adaptation to LOTOS

The reader should notice that the kripke model requires each state to have at least one successor because the transition relation $\Re$ is total. To deal with our behaviour trees, we have slightly generalized the model to allow $\Re$ to be partial. This is due to the following points:

(i) In LOTOS, the computations are not necessarily infinite. A process may deadlock (stop), or terminate successfully hence no transition is possible from that state.

(ii) SELA requires the user always to provide values for the boundaries of the tree, namely, the width (the maximum number of next actions for each node) and the depth (maximum number of successive multiple derivations) of the tree. If the symbolic tree of a specification exceeds these boundaries, SELA will leave it incomplete (section 2.4.4). This case was considered separately and model checking can proceed with this incomplete tree, but of course some properties can not be

checked completely. In this case we have to display an appropriate message stating that the property was verified on the portion of the tree that was generated.

This problem of infinite computations is solved by other authors [JKP89] by adding self loops to terminal states in order to have infinite paths. Terminal states in this case are extra states that are labelled *true*. We adapt the same algorithm introduced by [JKP89] (section 3.2.3) without adding self loops to terminal states. A terminal state in our case will have an empty successor set. We refer to our tree model as a kripke-like structure for this reason since our computations are not necessarily infinite. The states in our model are classified into three categories:

(i) *normal states* which have non empty successor sets. These states include the special initial state that is labelled *true*.

(ii) *terminal states* which have empty successor sets. No transition is possible from these states.

(iii) *continue states* which indicate that the maximum depth of the tree has been reached. Theses states cannot be treated in the same way as the terminal ones, because they may have successors if the maximum depth is incremented. For this reason, the successor set of a continue state is not determined.

In the next section, we introduce a modified satisfaction relation for CTL operators which takes into consideration total and partial transition relations.

4.4 A Modified Satisfaction Relation

We recall that in section 3.3.2, a definition of a computation path was given and the CTL satisfaction relation was outlined. However, neither the definition, nor the satisfaction relation, took into consideration finite computations. We need to modify slightly the definition to include finite sequences of transitions and to update the satisfaction relation accordingly. From now on, we will classify a computation path in one of the following categories:

(i) *complete computation paths* which can be finite sequences of states $(q_0, q_1, q_2,..,q_f)$ where q_f is a terminal state, or possibly infinite sequences of states $(q_0, q_1, q_2,..)$ such that:$(\forall\, q_i \in K)\ (\exists\, q_{i+1} \in K)\ [<q, q_{i+1}> \in \Re]$.

(ii) *continue computation paths* which terminate with a continue state indicating that they are incomplete.

Clearly, only the next time operators need to be modified. A next time formula is not satisfied at terminal states. We will use *terminal(s)* to denote that state *s* is a terminal state, we get:

$s \models AX(f)$ iff 1. $\neg$ terminal(s), and
2. $\forall\, s' \in Suc(s), s' \models f$.

$s \models EX(f)$ iff 1. $\neg$ terminal(s), and
2. $\exists\ s' \in Suc(s), s' \models f$.

□

However, it should be noticed that in the case of a continue state, we cannot conclude whether a next time formula holds or not because the path is not complete. Therefore that case is not included in the satisfaction relation.

The fixpoint characterization of each of the CTL operators discussed in section 4.1 consists of a recursive test for the truth value of *f*, and supposes that each state has a successor. It is obvious that with the modified satisfaction relation, these characterizations are not always valid since the transition relation is not necessarily total. We need to update the fixpoint representation for each of the operators. We will use the notation $OP_s(f)$ where *OP* stands for one of the operators (*AG*, *EG*, ...) to denote the proposition *f* prefixed by *OP* at state *s*, and *terminal*(*s*) to mean that *s* is a terminal state. The index is omitted when *OP* is preceded by a next time operator because the state is not determined at that point. The fixpoint characterization for each of the operators is given below:

$$
\begin{aligned}
AG_s(f) &\equiv f \wedge (terminal(s) \vee AX(AG(f))) \\
EG_s(f) &\equiv f \wedge (terminal(s) \vee EX(AG(f))) \\
AF_s(f) &\equiv f \vee (\neg terminal(s) \wedge AX(AF(f))) \\
EF_s(f) &\equiv f \vee (\neg terminal(s) \wedge EX(EF(f))) \\
A_s[f_1 \upsilon f_2] &\equiv f_2 \vee (\neg terminal(s) \wedge f_1 \wedge AX(A[f_1 \upsilon f_2])) \\
E_s[f_1 \upsilon f_2] &\equiv f_2 \vee (\neg terminal(s) \wedge f_1 \wedge EX(E[f_1 \upsilon f_2]))
\end{aligned}
$$

□

The algorithms discussed in chapter 4 were based on the original fixpoint characterization of the CTL operators. They need to be updated to take into account the modified satisfaction relation.

4.5 The Modified Algorithms

The original algorithms were defined over trees with infinite paths. This means that the exploration of a branch ends when a state is revisited, in which case a cycle has been detected, or when the immediate assertion of a formula fails at a particular state in that branch. The modified algorithms have to deal not only with infinite computations, but also with finite paths and partial trees. In other words, we need to generalize the algorithms for both complete and continue computation paths.

Recall that the boolean function *holds(f,s_i)* is set to true when the formula f is true at state s_i. The boolean flag *visited(s_i)* is set to true when state s_i is visited during the depth-first search. We will use also the boolean flag *cut* which will be set to true when a continue state is reached.

The algorithms for *AX(f)* and *EX(f)* at state s_i are very similar to the ones discussed in section 4.2.1 except that now, we have to check whether a state is a terminal or a continue state. A next time formula returns false when it is checked on a terminal state. In the case of a continue state, the satisfiability of the next time formula can not be determined. Implementing the algorithm implies that an appropriate message has to be displayed to the user stating that the satisfiability of the formula can not be determined for the specified depth.

We will use the operators *AG(f)* and $E[f_1 \upsilon f_2]$ to illustrate the modifications added to the algorithms. For the other operators, the algorithms are derived in the same fashion.

The additions to the algorithm For the operator *AG(f)* are needed after the immediate assertion of a formula f is found to be true. In the original algorithm, we proceed to check recursively all the immediate successors of the state. In this case, we also check whether it is a terminal or a continue state (lines 7 to 14). If the state s_i is terminal, then *holds(AG(f), s_i)* is set to true and the control is returned to the previous call (lines 7 to 9). If s_i is a continue state, we set *holds(AG(f), s_i)* to true and set the special flag *cut* to true. This flag will be used by a calling procedure to determine whether the formula is completely or partially satisfied on the model. After the algorithm terminates, three cases arise:

(i) *holds(AG(f), s_0)* returns false meaning that f was found to be false at some state s_i in the model which caused the algorithm to halt.

(ii) *holds(AG(f), s_0)* returns true and the flag *cut* was not set to true. This case yields to the conclusion that the formula is completely satisfied on the model.

(iii) *holds(AG(f), s_0)* returns true and the flag *cut* is true. This case means that the tree is cut up to a specified depth and that the formula was satisfied up to that depth. In other words, the formula is partially satisfied. The user may want to increase the depth of the tree (by reexecuting SELA) to gain more confidence in the result.

It is clear that with these modifications, we can determine the satisfiability of a formula in some cases without having to generate the whole tree. If a formula is not satisfied on a partial model, then it is not satisfied on the whole model. One can see immediately the advantage of such feature when the model is too large to fit in memory.

The algorithm for $AG(f)$ can be written as follows:

```
1.  if visited(s_i) then
2.          holds(AG(f),s_i) ← true
3.          return
4.  else
5.          visited(si) ← true
6.          if holds(f,s_i) then
7.              if terminal(s_i) then
8.                  holds(AG(f),s_i) ← true
9.                  return
10.             else
11.             if continue(s_i) then
12.                 holds(AG(f),s_i) ← true
13.                 cut ← true
14.                 return
15.             else
16.                 for all successors t of s_i such that (s_i,t) ∈ ℜ do
17.                     if ¬ holds(AG(f),t) then
18.                     holds(AG(f),s_i) ← false
19.                     return
20.                     endif
21.                 endfor
22.                 holds(AG(f),s_i) ← true
23.                 return
24.             endif
25.         else
26.             holds(AG(f),s_i) ← false
27.             return
28.         endif
29.  endif
```

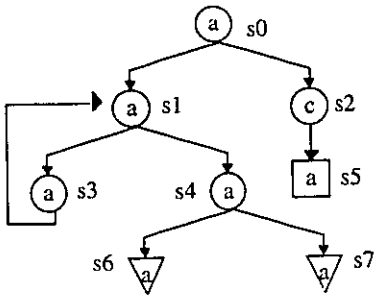
□

We will use graphically a square to represent terminal states, a circle to represent normal states, and a triangle to represent continue states.

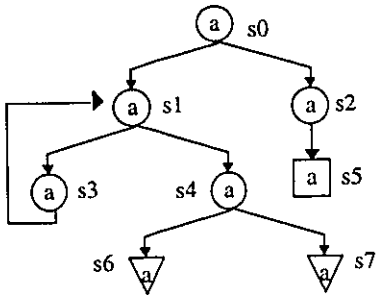
Example

The following tree is not a model of $AG(a)$. The presence of the two continue states (s_7 and s_8) does not affect the evaluation of the formula. The immediate assertion of a is

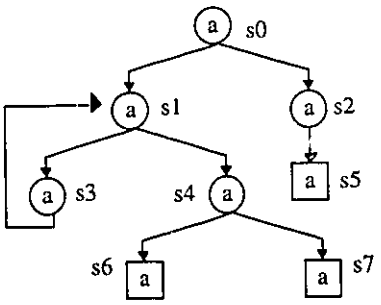
found to be false at state s_2 which is enough to conclude that *AG(a)* will not be satisfied even if we increase the depth of the tree.



We can conclude on the other hand that *EG(a)* holds on the model since there is a path that satisfies the formula $(s_0, s_1, s_3, s_1, s_3, ...)$. The formula *AG(a)* is satisfied on the following tree up to the specified depth:



The following tree is a complete model since it contains only full computation paths. Finite paths terminate at one of the states s_5 or s_6. The tree is in fact a model of *AG(a)*.



□

For the operator $E[f_1 \upsilon f_2]$, the additions to the algorithm are needed after the immediate assertion of f_1 (but not f_2) is found to be true at a certain state s_i. In the original algorithm, we proceed to check recursively all the immediate successors of the state. In this case, we also check whether it is a terminal or a continue state (lines 15 to 22). If the state s_i is terminal, then *holds*$(E[f_1 \upsilon f_2], s_i)$ is set to false and the control is returned to the previous call (lines 15 to 17). The function is set to false because the current state is terminal and therefore, there is no possible transition that will lead eventually to a state on which f_2 holds. If s_i is a continue state, we set *holds*$(E[f_1 \upsilon f_2], s_i)$ to false and set the special flag *cut* to true (lines 19 to 22). At that moment, we can not decide whether $E[f_1 \upsilon f_2]$ is satisfied or not because the tree is cut. Therefore, we set the function to false in order to force the algorithm to return to the pervious call and explore the other successors (if any). The flag *cut* will be used by a calling procedure to determine the satisfiability of the formula. It follows that three cases arise:

(i) *holds*$(E[f_1 \upsilon f_2], s_0)$ returns false and the flag *cut* is not set to true. this means that all the paths were covered, and for each path, a state s_i was reached such that f_1 is false, or the whole path was explored and f_2 was never reached.

(ii) *holds*$(E[f_1 \upsilon f_2], s_0)$ returns false and the flag *cut* is set to true. This case leads to the conclusion that no decision can be taken for the specified depth. The user has no choice but to increase the depth of the tree.

(iii) *holds*$(E[f_1 \upsilon f_2], s_0)$ returns true indicating that $E[f_1 \upsilon f_2]$ holds on the model. In other words, some state s_i where f_2 holds, was reached and f_1 holds for all the states that preceded it during the depth-first search. The boolean value of the flag *cut* does not affect the conclusion in this case since the formula is existential, and hence it takes at least one path that satisfies the formula to decide the satisfiability of the formula on the model.

The updated algorithm for $E[f_1 \upsilon f_2]$ can be written as follows:

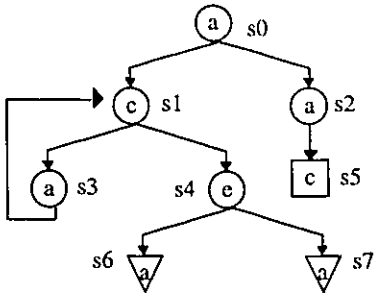
```
1.  if visited(s_i) then
2.          holds(E[f_1 υ f_2],s_i) ← false
3.          return
4.  else
5.          visited(s_i) ← true
6.          if holds(f_2,s_i) then
7.                  holds(E[f_1 υ f_2],s_i) ← true
8.                  return
9.          else
10.                 if ¬ holds(f_1,s_i) then
11.                         holds(E[f_1 υ f_2],s_i) ← false
12.                         return
13.                 endif
14.         endif
15.         if terminal(s_i) then
16.                 holds(E[f_1 υ f_2],s_i) ← false
17.                 return
18.         else
19.                 if continue(s_i) then
20.                         holds(E[f_1 υ f_2],s_i) ← false
21.                         cut ← true
22.                         return
23.         else
24.                 for all successors t of s_i such that (s_i,t) ∈ ℜ do
25.                         if holds(E[f_1 υ f_2],t) then
26.                         holds(E[f_1 υ f_2],s_i) ← true
27.                         return
28.                         endif
29.                 endfor
30.                 holds(E[f_1 υ f_2],s_i) ← false
31.                 return
32.         endif
33.  endif
```

□

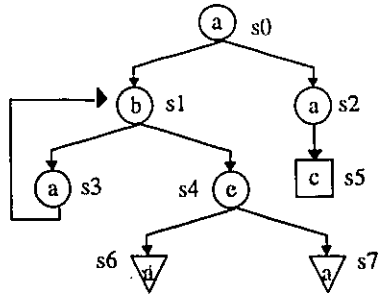
Example

Given the following tree, we can check easily that $A[a \upsilon c]$, *AF(c)* and *EF(e)* are satisfied. On the other hand, we cannot conclude whether *EF(b)* is satisfied or not

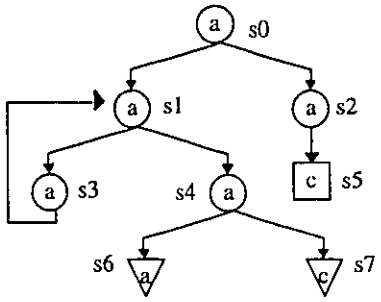
because the tree is cut and we need to increase the depth of the tree to make such conclusion.



For the following tree, we can check that $E[a \upsilon c]$ is satisfied and that $E[a \upsilon e]$ is not.



We cannot conclude for the specified depth of the following tree whether $E[a \upsilon e]$ is satisfied or not. Every state in the path (s_0, s_1, s_4, s_6) is labelled a and since this path terminates with a continue state (continue path), we need to increase the depth of the tree in order to determine the satisfiability of the formula.



□

4.6 Chapter Summary

We have presented in this chapter the algorithms used to implement our model checker. The algorithms are derived from [CES86]. However, only the "until" operator algorithm was explicitly discussed in this reference. We used the same approach and derived the checking algorithms for all the operators. We have also seen that the algorithms derived in this way cannot be applied immediately to LOTOS, since in LOTOS (contrary to the model discussed in [CES86]), computations are not necessarily infinite (section 4.3). For this purpose, a modified satisfaction relation for CTL was introduced. The fact that symbolic trees generated by SELA may be incomplete (cut to a maximum depth) motivated us to modify the algorithms further in order to make it possible to apply model checking even on partial trees.

Chapter 5

LMC Design and Implementation

We discuss in this chapter the practical issues related to the design and implementation of LMC. We will give the overall structure of LMC and discuss each of its modules separately. However, we will not describe full implementation details since we discussed in chapter 4 the algorithms used for that purpose. On the other hand, we will describe the graph model used by the model checker and the related data structures. Also, we will discuss the kind of properties handled by LMC and show how these properties can be formulated starting from the initial requirements.

5.1 General Structure of LMC

We recall that in section 1.3 we presented a design methodology that resulted naturally in the integration of our model checker with other tools such as ISLA and SELA. The methodology involved a scenario that starts from the informal requirements and ends up with the checking of the model representing a LOTOS specification against a set of correctness properties. LMC also uses the abstract data types interpreter, called SVELDA (System for Validating and Executing LOTOS Data Abstractions) [Feh87] to evaluate guards and selection predicates encountered during the model checking procedure.

LMC is programmed in Quintus Prolog under the Unix operating system. Prolog was chosen as the implementation language for the following reasons:

- Prolog supports dynamic allocation of facts and rules, unification, direct substitution and backtracking. These are all of great importance for the implementation of LMC.

- Both the model checking algorithms and the resolution algorithms applied by Prolog perform a depth-first search of a given data structure; the data structure being a state graph in the case of the model checking algorithms and the facts of a Prolog program in the case of Prolog's resolution algorithms. By implementing LMC in Prolog, we took advantage of its built-in resolution mechanisms.

- The nature of Prolog makes it suitable for the development of individual functionalities which can be easily added, tested, modified, or removed.

- The internal representation of the model used by the model checker is a Prolog data base. Also, most of the tools of The LOTOS "toolkit" developed at the University of Ottawa and described in section 1.3 are implemented in Prolog.

□

LMC consists of three main modules:

(i) the interface module

(ii) the transformer

(iii) the model checker

Each of these modules has its main functionalities. Figure 5.1 shows the overall structure of LMC and the relations between its components and with other tools. A general description of each of the modules and their functionalities will be discussed in the next sections.

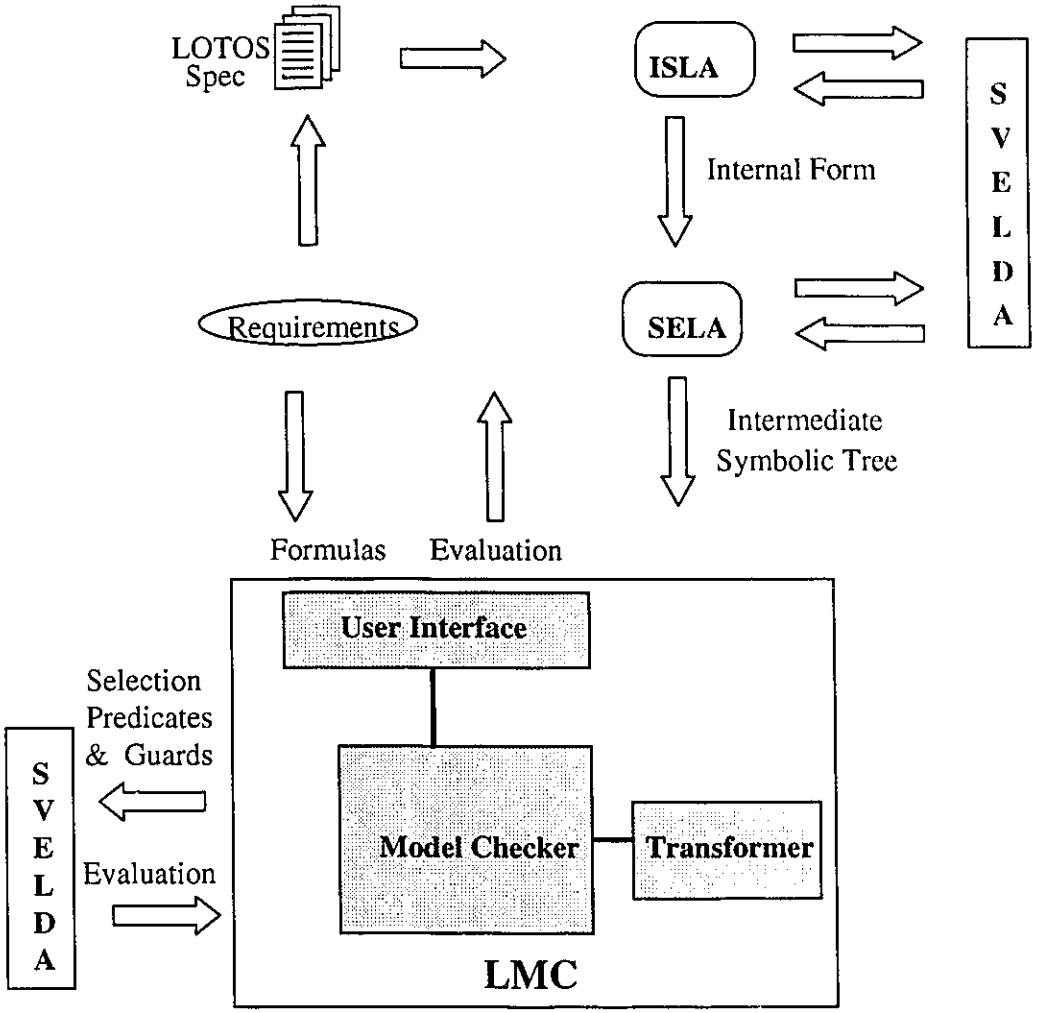


Figure 5.1 Overall Structure of LMC

5.2 The Interface Module

The role of the interface module is to provide a means for the user to interact with the model checker via a user friendly environment for the verification of LOTOS specifications. The main functions provided by the interface module can be summarized by the following points:

- Provide the user with menus that will allow him to interact with the model checker.

- Parse the formulas entered by the user since a formula can be subject to syntax errors (for example, the user might enter an extra parenthesis, a wrong operator or even a gate name that is not defined in the LOTOS specification).

5.3 The Transformer

We presented in section 2.4 the symbolic expansion of LOTOS specifications using the symbolic expander SELA. In fact, the symbolic behaviour tree generated by SELA is not used directly by the model checker, it represents an intermediate symbolic behaviour tree that can be used for other validation purposes such as finding test cases for the specification [Ash92], or deriving the canonical tester of the specification in the case of conformance testing [Jaou92]. The reason behind this is that the behaviour tree generated by SELA has labelled transitions rather than labelled states as opposed to the semantics of CTL which are defined over labelled state trees (kripke structures). Therefore, a simple transformation is needed to map the intermediate symbolic tree generated by SELA into a kripke-like structure (section 4.3). This transformation is a direct application of the algorithm described in section 3.2.3 with the only exception that we do not add self loops to terminal states. A terminal state in our case will be characterized by an empty successor set.

5.3.1 The Tree Model

The model used by the model checker is a symbolic behaviour tree described by:

(i) An initial state.

(ii) A finite set of states labelled with the following information:

- A guard list (possibly empty), which is a precondition that, if evaluated to true, leads to enabling the action at the current state and all its successors.
- A gate name, which can be visible or internal. A visible action must be declared in the formal gate list of the specification. An internal action can be the explicit action "*i*" or an action on a gate hidden from the environment by the "*hide*" operator.

- An experiment list (possibly empty). An experiment can be either an offer "!" of a symbolic value or an offer to accept a value "?"
- A predicate list (possibly empty), which is a post condition established on the values that can be accepted/offered.
- The set of all the successors which can be possibly empty when the state is terminal.

(iii) A transition relation between the states.

□

5.3.2 An Example

Consider the LOTOS behaviour given by:

```
1.    hide g1 in g1?x:nat[x > 2];
2.    g2?y:nat;
3.    ([x = y] -> g2 ! x;stop
4.    []
5.    g3?y:nat[y = 0];
6.    exit(y))
```

SELA generates the following intermediate symbolic behaviour tree:

```
bh0 * 1 i (hiding: g1 ?nat@1:Nat [nat@1 > 2] ) [1]
bh1 * | 1 g2 ?nat@2:nat [2]
bh2 * | | 1 [nat@1 = nat@2] g2 !nat@1 [3] Deadlock
    * | | 2 g3 ?nat@3:nat [nat@3 = 0]  [5]
bh4 * | | | 1 exit !nat@3 [6]
```

The intermediate symbolic tree is in fact generated simultaneously in the external form shown above and in an internal form, which already contains the information necessary for further transformation. The internal representation is a Prolog database which consists of nodes in the tree and actions corresponding to the edges between the nodes. Figure 5.2 represents the tree model of the example obtained after applying the transformation:

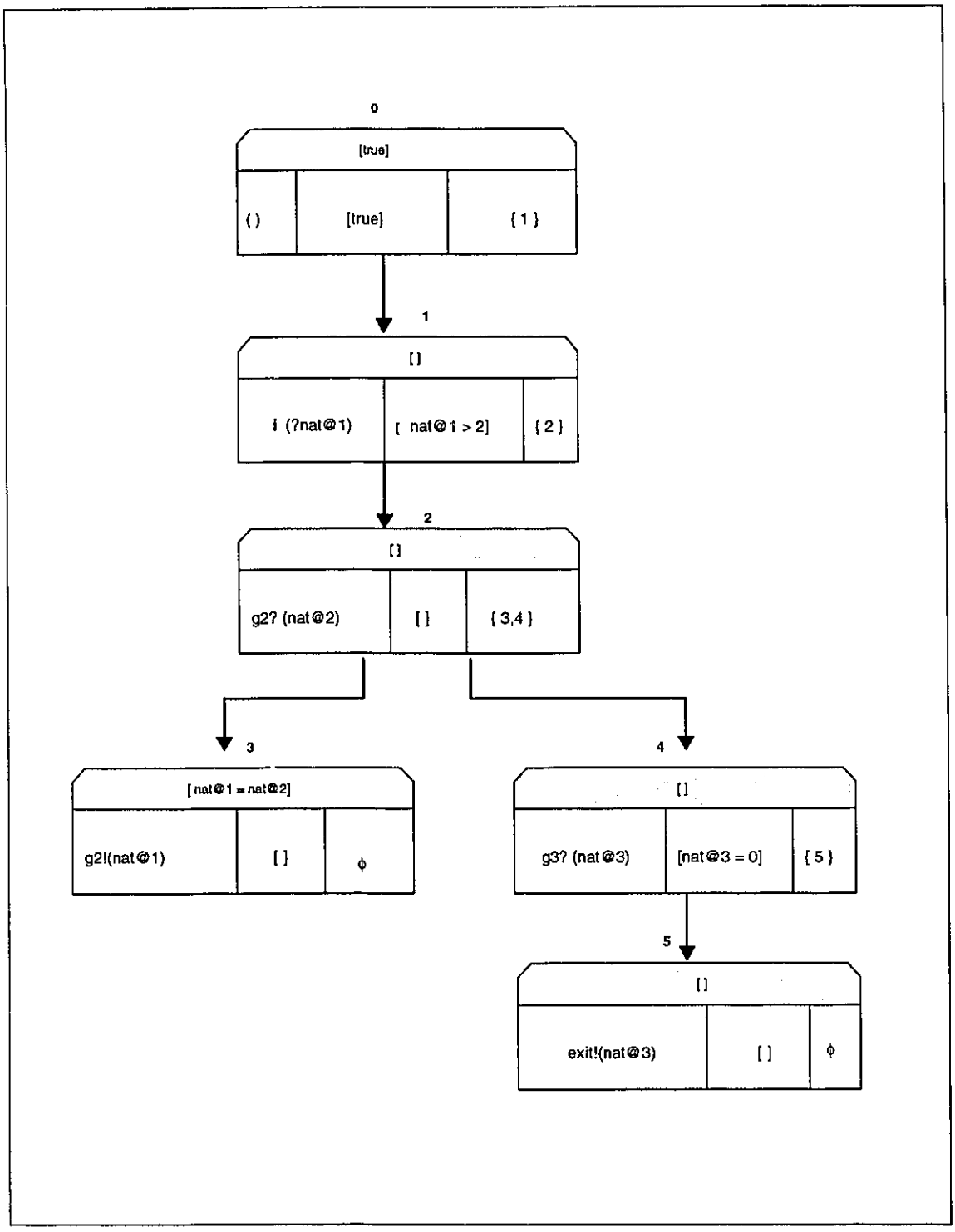


Figure 5.2 tree model used by the model checker

Table 5.1 summarizes the components of each state in the previous graph :

	Guard	Action	Experiment List	Predicate	Successors
State 0	[true]	( )	[ ]	[true]	{ 1 }
State 1	[ ]	i (g1)	[?nat@1]	[nat@1 > 2]	{ 2 }
State 2	[ ]	g2	[?nat@2]	[ ]	{ 3, 4 }
State3	[nat@1 = nat@2]	g2	[!nat@1]	[ ]	∅
State 4	[ ]	g3	[?nat@3]	[nat@3 = 0]	{ 5 }
State 5	[ ]	exit	[!nat@3]	[ ]	∅

Table 5.1 Components of the states

It is important to notice the following aspects:

- An *exit* is considered as an extra observable action when the specification (or process) is declared to exit a value. In the example, the successful termination *exit(y)* is represented by an offer experiment "*!nat@3*".

- A *stop* in the original specification indicates an empty successor set. States 3 and 5 are terminal states.

- Variable instances are replaced by their symbolic values. For instance, the value of *y* is *nat@2* in state 2 and *nat@3* in states 4 and 5. The symbolic values must be unique so that scopes and bindings of variables don't change when using the expansion rules (section 2.4.5).

- A gate that is hidden is replaced by the internal action *i*.

□

5.4 The Model Checker

The model checker is an implementation of the CTL algorithms outlined in chapter 4. It determines whether a correctness property is satisfied or not on the model corresponding

to a given LOTOS specification. To do so, the model checker visits the states of the model and attempts to match the actions of the formulas with those of the states according to the types of interactions between processes in LOTOS. This method can be considered somehow similar to using the method of "testing processes" which consists of creating processes to run in parallel with the specification [PL91, Bou91]. Such processes contain simple sequences of actions with value parameters that will synchronize with the external actions of the specification. In our case, the testing processes correspond to the temporal logic formulas which are a combination of CTL operators and LOTOS actions. The value parameters specified in the formulas will synchronize with particular actions of the specification according to the temporal operators.

The model checker assists the interface module by saving information during the model checking process. This information consists of the list of visited states during the evaluation up to where a formula fails or succeeds, and will be available to the user via the interface module. Next, we present the syntax of the formulas accepted by the model checker.

5.4.1 Formulation of the Properties

The formulas accepted by the model checker are in fact, a combination of the CTL operators and of LOTOS actions that include only offer experiments. The user can inject the values to be provided by the environment into the model representing the specification by supplying them within the formulas. Values that can be generated internally by the specification without the intervention of the environment can also be simulated.

Let p denote the basic component of a formula. It can be either a gate name associated with a list of offer events or an exit associated with values. p can be written as follows:

$$gate\_name\ Exp_1\ Exp_2 \ldots Exp_n \quad 0 \leq i \text{ or}$$
$$exit(Value_1, Value_2, \ldots, Value_n)$$

where:

gate_name is a gate name that is declared in the formal gate list of the specification or is hidden using the "*hide*" operator.

Exp_i is an offer experiment of the form "*! Value*$_i$" and $Value_i$ represents a value and falls in one of the following three categories:

(i) An explicit value that can be generated internally by the specification such as the sequence number associated with messages in the case of the alternating bit protocol.

(ii) A symbolic value representing values provided by the environment represented by @1, @2,...

(iii) A "don't care" value, represented by a "*".

By composing these basic components using CTL operators, we can formulate the initial requirements of the system specified and use LMC to validate them. For example, the following formula :

$$AG(send!* \Rightarrow AF(receive!*))$$

means that whenever something is sent, something will be received eventually. One can also write :

$$AG(send!succ(0) \Rightarrow EF(receive!succ(0)))$$

which means that whenever the value *succ(0)* is sent, there is some sequence leading to receiving that value in the future.

Note that "*" is not the same as "?" because in the case of a "?" a variable becomes bound to a value, while in the case of "*" we mean that the value of the variable is not important.

5.4.2 The Synchronization Procedure

Recall that in section 2.3, we presented the types of interaction between LOTOS processes (table 2.1). These types of interaction occur when two or more processes having a "rendez-vous" on a gate, agree on one or more values to be established. There are mainly three types of interaction:

- *value matching* which happens when both processes are offering value expressions. Both value expressions have to be evaluated to the same value in order for the synchronization to occur.

- *value passing* which occurs when a process is offering a value expression while the other process is ready to accept a value. The synchronization occurs when the value offered by the first process belongs to the sort of the expected value by the second process.

- *value generation* which happens when both processes are ready to accept values. The values to be accepted must be of the same sort in order for the synchronization to occur.

Formulas accepted by the model checker are based on the first two types of interactions. The user plays the role of the environment and therefore, he/she has to supply values to allow the synchronization to occur. We will give next, some examples of interactions that might occur during the model checking procedure.

Example

From section 5.4.1, values specified in the formulas can be explicit, symbolic or even replaced by a "*" to mean that they are not important. The following table contains some examples of possible interactions during the model checking procedure.

Case	State Actions	Formulas Actions	Result
1	get?nat@1	get!0	nat@1 = 0
2	get?nat@1 [nat@1 < 5]	get!7	no synchronization
3	get?nat@1 [nat@1 < 5]	get!@3	nat@1 = @3 & @1 < 5
4	get?nat@1	get!*	nat@1 = * ("don't care" value)
5	get?nat@1 [nat@1 < 5]	get!*	nat@1 = *
6	get?nat@1	give!0	no synchronization
7	give!0	give!succ(0)	no synchronization
8	give!nat@1	give!0	synchronization possible
9	give!nat@1	give!*	nat@1 = *

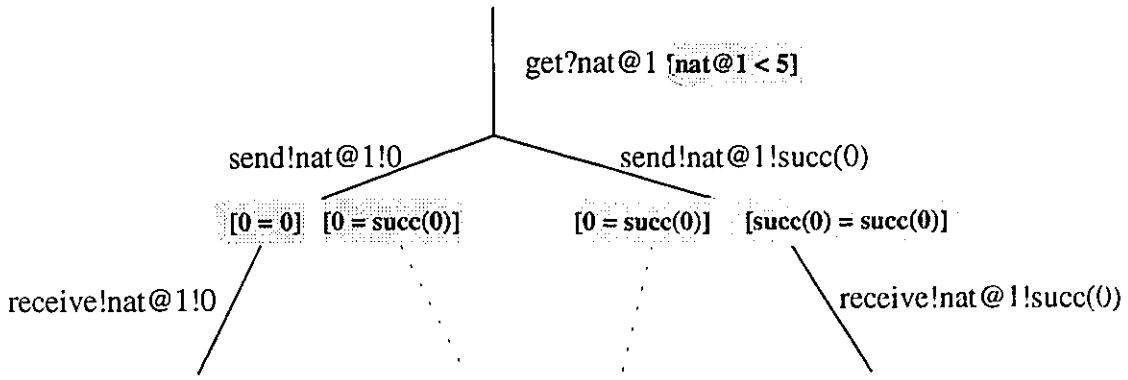
Synchronization is impossible in case *2* since the value *7* does not satisfy the restricting condition. In case 5, the restricting condition is neglected since the value offered is a "*". In case 8, synchronization occurs only if *nat@1* was previously assigned the value *0*.

5.4.3 Evaluation of Selection Predicates and Guards

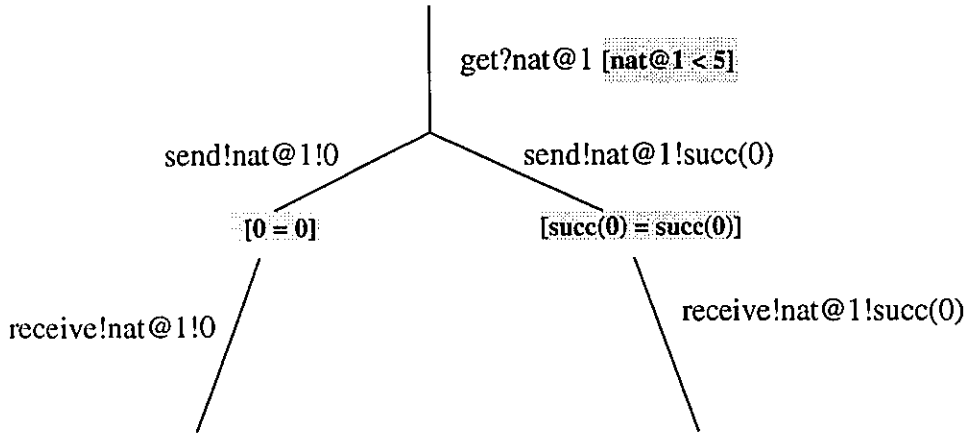
During the model checking procedure, variables in the symbolic tree are instantiated due to the synchronization with the actions of the formulas. Therefore, various execution sequences in the tree can be explored in order to determine the satisfiability of the formulas. During the exploration of such execution sequences, it is possible to encounter selection predicates and guards that have to be evaluated in order to determine the feasibility of the sequences. In fact, during the transformation of LOTOS specifications to behaviour trees using SELA, guards and selection predicates that are evaluated to be always true are kept in the corresponding branches whereas those that are always false are eliminated and so are the branches that follow them. SELA does not prune a branch when the guards or predicates cannot be evaluated.

Example

During the transformation process, SELA generates an intermediate symbolic tree with labelling over the edges. We will suppose that at some point, during the transformation of a certain LOTOS specification using SELA, the following tree is obtained:



This behaviour tree represents a process that accepts a natural number represented symbolically by *nat@1* at gate *get*. This action is followed by a selection predicate that restricts the value of *nat@1* to be less than *5*. Then that number is sent via gate *send* along with a sequence number *0* or *succ(0)*. Let's suppose also that the number is received at gate *receive* only if the sequence number is equal to a certain expected sequence number and that SELA detects four branches where two of them will never occur because their corresponding guards are always false ([*0* = *succ(0)*]). In that case, the corresponding branches (represented in the tree by dotted lines) are not needed and therefore, they are pruned. The tree will then contain only the selection predicates and guards that are always true and those that cannot be evaluated.



□

The main problem using symbolic trees for model checking is that many of the branches contain selection predicates and guards that can not be evaluated because some of their parameters were not instantiated in the formulas.

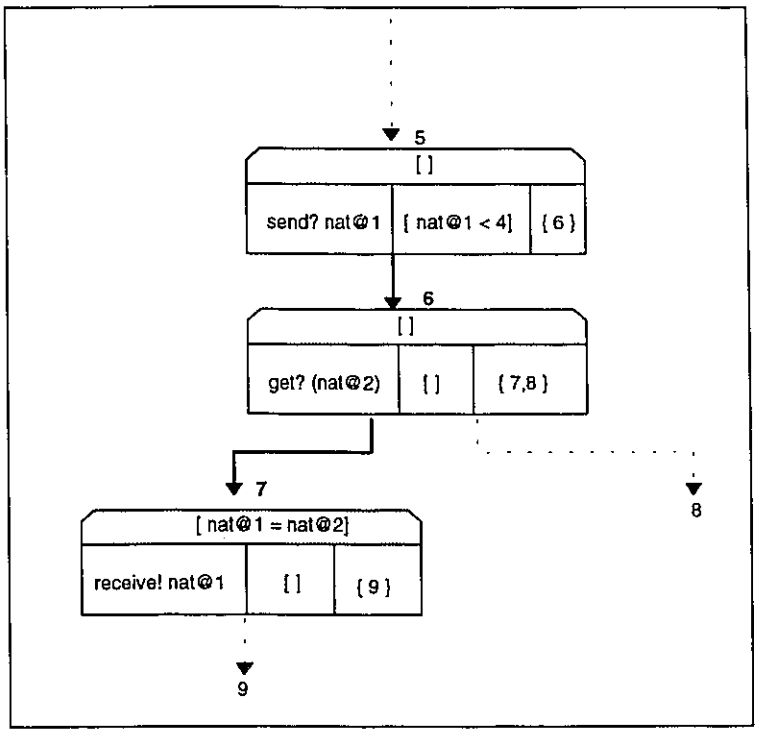
Example

Suppose we want to prove the satisfiability of the following formula:

$$AG(send!0 \Rightarrow AF(receive!0))$$

which states that whenever the value *0* is sent at gate *send* then every sequence must lead eventually to receiving *0*.

The formula is preceded by the universal global operator; this means for every state in the tree in which synchronization with *send!0* occurs, we have to explore every possible future sequence starting from that state in order to reach a state in which synchronization with *receive!0* is possible. Suppose that during the model checking procedure, we reach the following subtree:



Consider for instance the sequence of states {5,6,7}. First the synchronization between *send?nat@1* and *send!0* occurs since *0* is a natural number and it is less than *4*. Then the search for a state in which synchronization with *receive!0* is possible starts. Synchronization fails at state 6, therefore the exploration of the next states proceeds. The guard [*nat@1* = *nat@2*] at state *7* is equivalent to [*0* = *nat@2*] and therefore cannot be evaluated since *nat@2* was not instantiated. This is because there was no synchronization at state *6*, and the only way to avoid this problem is to execute the specification in a step-by-step mode so that all parameters can be assigned values in an interactive mode. In our

model checker, we handle such situation by assuming that the condition is true and model checking proceeds.

□

The abstract data types interpreter SVELDA is able to evaluate selection predicates and guards provided that all the parameters are assigned values. Since this is not the case for model checking, we had to take some assumptions into considerations. These assumptions can be summarized in the following cases:

- All the parameters are assigned values, in which case evaluation can proceed.

- Some parameters are assigned the don't care value "*", in which case the corresponding expressions are evaluated to true since the actual values are not important. For example, the guard [*odd(nat@1)*] is evaluated to true if *nat@1* was assigned the value "*" while the guard [*odd(nat@1) and nat@2 > 3*] is evaluated to false if *nat@1* was assigned the value "*" and *nat@2* the value *1*.

- Some parameters are not assigned values for the reasons explained in the example above, in which case the corresponding expressions are assumed to be true. For example, the selection predicate [*0 + coin@1 + coin@2 >= price(button_name@1)*] is supposed to be true if any of the symbolic values is unknown.

The possible existence of unfeasible paths in models computed according to our method is a problem of which the user must be aware. These paths are unfeasible because they contain contradictory conditions. For example, by combining the selection predicates of several actions of a certain branch in the tree, we would get contradictory conditions such as [*ConReq(x) and ConInd(x)*] that requires *x* to be a connection request and a connection indication at the same time. Since *x* is used symbolically (i.e. no value is assigned to it), such a condition unfortunately cannot be evaluated; thus by our model checker it is assumed to be true. This fact causes some formulas to be evaluated to true when they should be evaluated to false. One way of solving this problem is to apply narrowing [RKKKL85] or theorem provers to predicates containing symbolic values. Another way to avoid unfeasible paths is to compose the specification in parallel with a process that provides all the necessary values, or insert these directly in the specification. Yet another

way would be to make available to the user in a tree form the list of all predicates assumed to be true. In this way, the user could see the contradictions by inspection.

This discussion shows the importance of appropriate diagnostics to accompany a model checker. Unfortunately, a diagnostic system is beyond what we are able to do in this thesis. However, the information produced by the model checker can be accessed in order to make it possible to follow the execution history up to the point where a formula fails or succeeds for diagnostics purposes. This is discussed in greater detail in section 5.6.

5.5 Properties Handled by LMC

In section 3.4, we discussed the main categories of the correctness properties, and how CTL can be applied to express these properties in general. This section identifies the types of properties that LMC handles with respect to our tree model.

The correctness properties that we address include:

Absence of deadlock

A deadlock occurs in a concurrent system if no process can progress. This case can be detected in our tree model by checking the presence of terminal states. Deadlock occurs in LOTOS when two or more processes fail to synchronize, in which case no transition is possible after that, or when the specification contains explicitly the inaction operator *stop*. Also, in the case of successful termination (section 2.3.7), the behaviour becomes equivalent to a *stop* after exiting which leads to considering it as a particular case of deadlock. Note that a process that terminates successfully may enable another process, in which case there is no deadlock. If we let S denote the set of states in the tree model, we can define a deadlock as follows:

$$deadlock \equiv \exists\, s \in S, \text{ such that } terminal(s).$$

This definition allows us to check easily the absence of deadlock in a specification once the corresponding tree model is generated. However, it does not allow us to specify explicitly in a formula the presence of *deadlock*. For this reason, we will consider

deadlock as an extra predicate that can be satisfied at a particular state. A state s that is deadlocked can be formally defined as follows:

$$s \models \textit{deadlock} \text{ iff } \textit{terminal}(s).$$

The property expressing the absence of deadlock in the whole specification can be written then as follows:

$$AG(\neg \textit{deadlock}).$$

If we want to check whether a particular action a leads always to a deadlock, we can write:

$$AG(a \rightarrow AF(\textit{deadlock})).$$

Absence of starvation

This case includes *guaranteed accessibility* and *responsiveness* properties. These properties usually express that requests for particular services lead to a response in the form of a granting of access. If we consider a process P (Figure 5.3) that communicates with its environment via two gates *send* and *receive*. A message m is delivered to the environment through gate *send* and acknowledgments *ack* are received through gate *receive*. A property stating for example that for every message delivered, an acknowledgment must be received, can be written as follows:

$$AG(send!m \rightarrow \; AF(receive!ack))$$

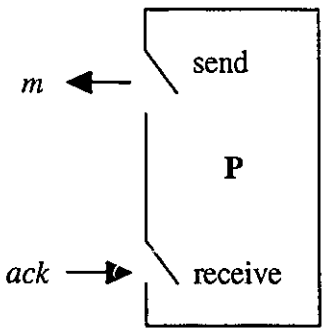


Figure 5.3 Process P

Precedence properties

These properties deal with the temporal ordering of events. By using the "until" operator, we can express properties of this type. For example, for process P described above, we can express the fact that whenever a message m_1 is sent, the process has to wait for the appropriate acknowledgment ack_1 before it delivers another message m_2 as follows:

$$AG(send!m_1 \rightarrow AX(A[\neg send!m_2 \cup receive!ack_1]))$$

□

5.6 Diagnostics

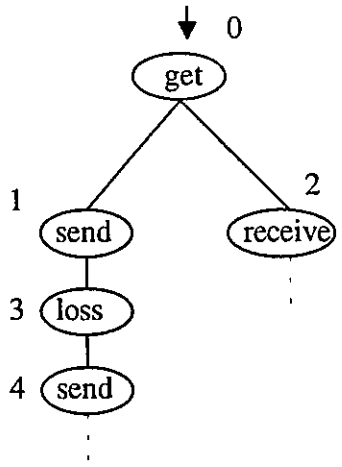
A model checker usually verifies whether properties expressed in temporal logic are satisfied on a given model. However, it does not explain why a certain formula fails or succeeds. For this reason, appropriate diagnostics must accompany a model checker in order to help the user detect erratic behaviours. Previous work has been done on this subject and can be found in [Ras90]. In this work, automated diagnostics can be performed based on sequences extracted from the model. In the case of complex formulas, the explanation can be refined (i.e. a higher level formula is explained in terms of its sub-formulas).

LMC is not capable of performing such functionalities for the moment. However, it allows the user to access the information produced by the model checker in order to make it possible to follow the execution history up to the point where a formula fails or succeeds. This information consists of the list of states visited during the model checking procedure. Another way to solve the problem is to invoke the interpreter ISLA based on the results of the evaluation of the formulas. In this case, using a step-by-step execution can help understand why a formula is not satisfied. It is important to notice that at this stage, the user has fewer alternatives to choose from since he knows what particular actions he wants to reach. As an example, he might be interested in checking if a data item received by a certain process will be delivered eventually to the environment. Suppose that the behaviour of the system under verification provides several choices along with the receiving action. It is obvious that the user is only interested in the paths that start with a *receive*. All the remaining alternatives are dropped and this reduces the

number of paths to be explored. This method is not efficient if the size of the model is large.

Example

Given the following simple model:



Suppose that we want to prove the following formula:

$$AG(send \rightarrow A[\neg send \ \upsilon \ receive])$$

which states that whenever a *send* is performed, no new *send* should be performed until there is a *receive*. It is obvious that this formula is not satisfied in the model since a new *send* can be performed in case of *loss*. This is illustrated in the sequence [*1*,*3*,*4*]. Since the main operator in the formula is global, the evaluation halts once a counter-example is found. The file generated by LMC contains the list of states visited up to the point where the formula fails. In the example, the file will contain the list [*0*,*1*,*3*,*4*]. Notice that state *2* does not belong to the list because the model checking algorithms are based on a depth-first search and therefore it had not been visited yet when the evaluation halted.

In a step-by-step mode, ISLA determines the set of next possible actions, based on the execution of the action provided by the user. However the user can drop some alternatives since the goal here is to follow the sequences imposed by the formula. For

instance, after executing the action *get* in the example, the user may choose between *send* and *receive*. Since we are interested in what happens when there is a *send*, then we can choose to explore the corresponding branch. This discussion shows that using an interpreter for diagnostics, can be less tedious than using it for simulation purposes because in the case of diagnostics, the user is guided by the formulas he wants to prove. If the formula is quite complex and the model is large enough, then invoking an interpreter or saving the list of states visited are not recommended. We are investigating new ways to provide automated diagnostics without the intervention of the user, based on sequences extracted from the model.

We give a more realistic example in appendix C of the way of operating presented above.

Chapter 6

Case Studies

In this chapter, we will use two examples to illustrate the validation methodology. The first example concerns a simple data link service provider and the second deals with a transport service handler. We start by giving an informal description of the desired service, then show how LOTOS is used to specify these requirements formally. The symbolic tree obtained is provided and finally examples of properties that can be verified are given. We do not discuss however the intermediate step involving the interpreter ISLA since it is used in our case only to check the syntax and static semantics of a LOTOS specification, and to generate an internal form useful for the symbolic expander SELA.

6.1 Example 1: A Data Link Service Provider

This example was first introduced in [QFP88]. It consists of a simple, one directional data link service provider (Figure 1).

6.1.1 Informal Description of the Service Provider

A transmitter is connected through a semi-duplex lossy line to a receiver. The alternating bit protocol is used for error detection. This means that every message sent by the transmitter is associated with a sequence number 0 or 1.

The transmitter: When a data item is provided by the environment to the *transmitter* through gate "*get*", the latter one sends it to process *line* along with a sequence number. A timer is set while the *transmitter* is waiting for an acknowledgment. Should a timeout occur before receiving the acknowledgment, the *transmitter* has no choice but to retransmit the original data. An acknowledgment with the wrong sequence number is ignored.

The receiver: The *receiver* is ready to receive a data item from process *line*, together with its sequence number. If the sequence number is as expected, the data item is delivered to the environment through gate "*give*" and an acknowledgment is sent with the sequence number of the next set of data. Otherwise, if the data item is old (having a wrong sequence number), an acknowledgment is sent to process *line* with the expected sequence number.

The line: Process *line* synchronizes on gate "*send*" with process *transmitter* when the message is of type "*info*", and with process *receiver* when the message is of type "*ack*". The synchronization works the opposite way for gate "*receive*".

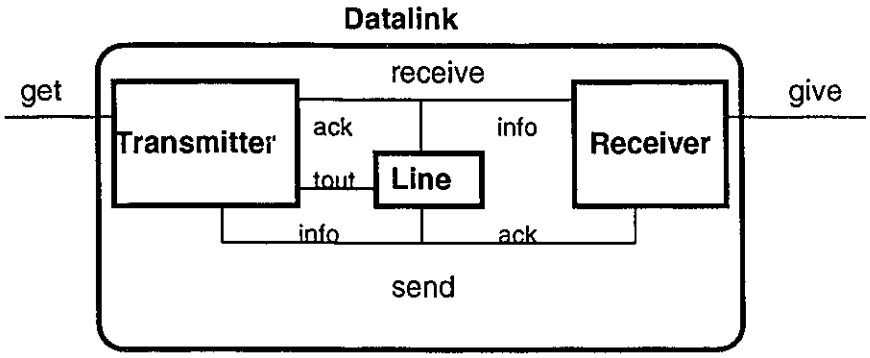


Figure 6.1 The Data Link Service Provider

6.1.2 The LOTOS Specification

We give here the main behaviour of the specification. The whole specification is provided in Appendix A.

Since the gates *tout*, *send*, and *receive* are not used to communicate with the environment, we hide them by using the *hide* operator. Both the *transmitter* and the *receiver* are executing in parallel independently (interleaving) but they have to synchronize with the *line* on the gates *tout*, *send* and *receive*. This is because the *line* is semi-duplex and therefore, we have to ensure that the *line* is accessed only by the *transmitter* when the latter one needs to send a data item or receive an acknowledgment and by the *receiver* when it needs to receive a data item or send an acknowledgment. This is described by the following behaviour:

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

Both the transmitter and the receiver are initialized with the same sequence number *0*.

6.1.3 The Symbolic Tree

After generating the symbolic tree and transforming it, we get the tree represented in figure 5. It was possible to represent the service with *31* states and *38* transitions. The edges are unidirectional from the top to the bottom of the tree except for those representing cycles. The information stored in the states is represented in the table below. The intermediate symbolic tree generated by SELA is provided in appendix A.

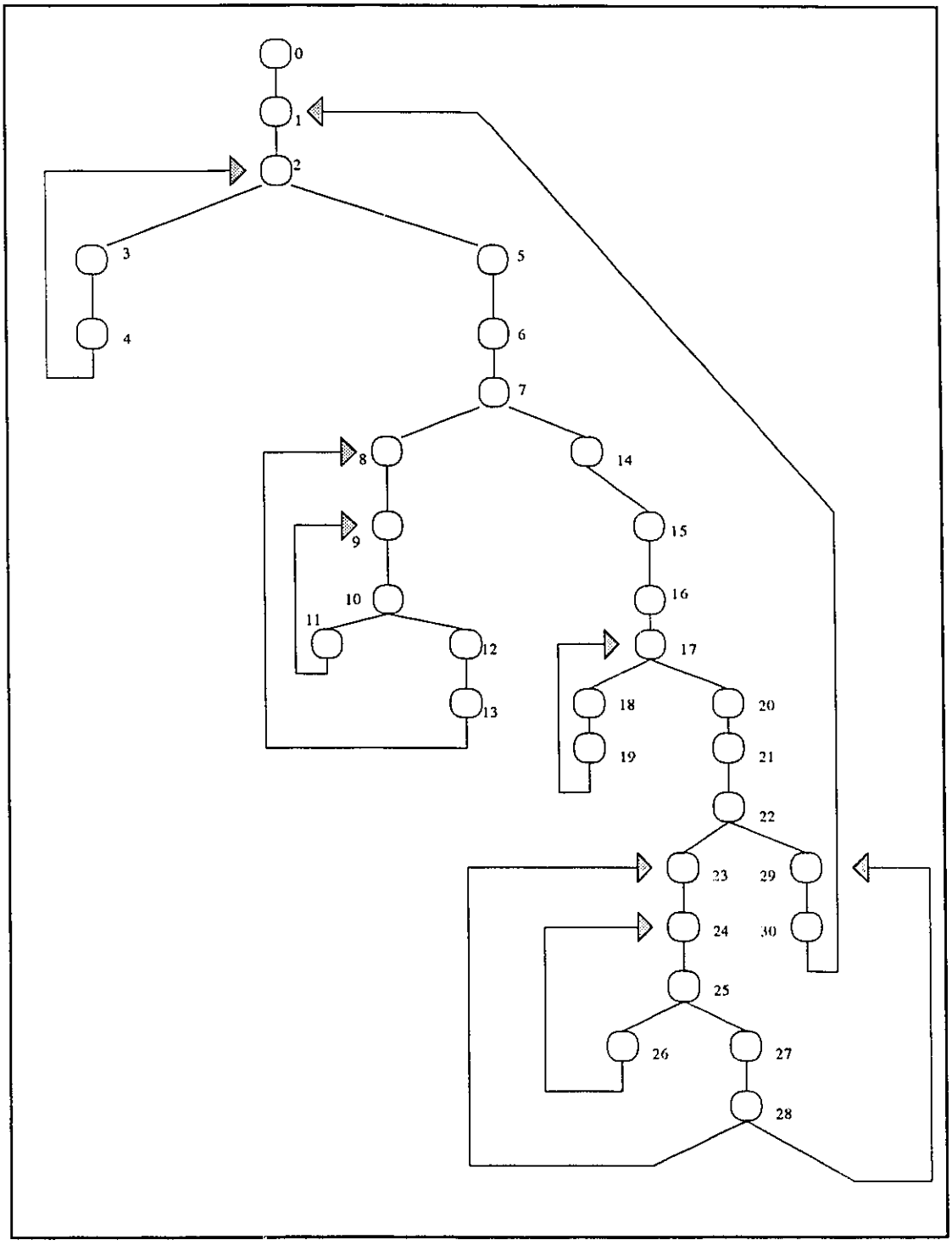


Figure 6.2 Symbolic tree for the datalink service provider

	Guard	Action	Experiment list	Predicate	Successors
State 0	[true]	( )	[]	[true]	{1}
State 1	[]	get	[?BitS@1]	[]	{2}
State 2	[]	send	[!info,!0,!BitS@1]	[]	{3,5}
State 3	[]	i	[]	-	{4}
State 4	[]	i(tout)	[]	[]	{2}
State 5	[]	receive	[!info,!0,!BitS@1]	[]	{6}
State 6	[0 = 0]	give	[!BitS@1]	[]	{7}
State 7	[]	send	[!ack,!inc(0),!empty]	[]	{8,14}
State 8	[]	i	[]	-	{9}
State 9	[]	i(tout)	[]	[]	{10}
State 10	[]	send	[!info,!0,!BitS@1]	[]	{11,12}
State 11	[]	i	[]	-	{9}
State 12	[]	receive	[!info,!0,!BitS@1]	[]	{13}
State 13	[inc(0) = inc(0)]	send	[!ack,!inc(0),!empty]	[]	{8}
State 14	[]	receive	[!ack,!inc(0),!empty]	[]	{15}
State 15	[]	δ	[]	[]	{16}
State 16	[]	get	[?BitS@1]	[]	{17}
State 17	[]	send	[!info,!inc(0),!BitS@1]	[]	{18,20}
State 18	[]	i	[]	-	{19}
State 19	[]	i(tout)	[]	[]	{17}
State 20	[]	receive	[!info,!inc(0),!BitS@1]	[]	{21}
State 21	[inc(0) = inc(0)]	give	[!BitS@1]	[]	{22}
State 22	[]	send	[!ack,!0,!empty]	[]	{23,29}
State 23	[]	i	[]	-	{24}
State 24	[]	i(tout)	[]	[]	{25}
State 25	[]	send	[!info,!inc(0),!BitS@1]	[]	{26,27}
State 26	[]	i	[]	-	{24}
State 27	[]	receive	[!info,!inc(0),!BitS@1]	[]	{28}
State 28	[inc(inc(0)=inc(inc(0)]	send	[!ack,!0,!empty]	[]	{23,29}
State 29	[]	receive	[!ack,!0,!empty]	[]	{30}
State 30	[]	δ	[]	[]	{1}

The guards in states 6 and 21 correspond to the case where the receiver receives a sequence number equal to the expected one (respectively 0 and 1) in which case the data item is delivered to the environment via gate give. States 13 and 28 represent the case where the received sequence number is not as expected. The expected sequence number at state 13 is 1 but the receiver receives 0 instead which leads to sending an acknowledgment with the expected sequence number. An acknowledgment with sequence 0 is sent at state 28.

6.1.4 Verification Phase

The original requirements are translated to a set of temporal logic formulas to be verified. The following are some examples of formulas that can be verified using LMC:

1) Absence of deadlock.

$$AG(\neg deadlock)$$

2) "if the data item *@1* is offered at gate *get* to process transmitter, then it is delivered to the environment at gate *give* in some future path by the receiver."

$$AG((get!@1) \Rightarrow EF(give!@1))$$

The use of the symbolic value *@1* makes it possible to express the fact that any particular data item provided to the datalink by the environment will be delivered without having to deal with all the possible values that represent messages.

The user may wish a stronger property to be true, i.e. that the data item is delivered in *all* future paths. Unfortunately this property cannot be true, because the medium can refuse forever to deliver an item. In the specification, this can be seen by the fact that infinite loops are possible, i.e. a timeout can occur every time a data item is sent by the transmitter. This is represented in the model of figure 6.2 by the sequence of states *2,3,4* which can be repeated infinitely. Delivery in all future paths could be guaranteed by a fairness assumption [CES86, Fran86]. We have no such assumption in the version of CTL that we have adopted.

3) If we are not interested in the data item delivered, but we simply want to verify that the sequence of actions is correct, then we might check the following property:

"if something is offered at gate *get* to process transmitter, then something is delivered to the environment at gate *give* in some future path by the receiver."

$$AG((get!*) \Rightarrow EF(give!*))$$

4) "No new *receive* operation after a *receive*, until a new *send* is performed."

$$AG((receive!*!*!*) \Rightarrow AX(A[\ \sim(receive!*!*!*) \cup (send!*!*!*)]))$$

5) If the receiver sends an acknowledgment with the sequence number *0* (i.e. it is expecting a message with the sequence number *0*) then one of the two following events should happen eventually:

- The receiver receives an info message with the wrong sequence number (*inc(0)*) in which case it should immediately send again the same acknowledgment with the expected sequence number (*0*).
- The receiver receives an info message with the expected sequence number (*0*).

$$AG(send!ack!0!empty \Rightarrow AF((receive!info!inc(0)!* \wedge AX(send!ack!0!empty)) \vee (receive!info!0!*)))$$

Similarly, we can prove a corresponding property when the receiver is expecting a message with the sequence number *inc(0)*.

6) "Whenever the transmitter sends a message of type info, it won't send another message of type info until the receiver receives the first message."

$$AG((send!info!*!*) \Rightarrow AX(A[\sim(send!info!*!*) \cup (receive!info!*!*))]))$$

This property obviously violates the protocol requirements because process transmitter can send again the same message of type info in case of timeout. It will be refused by the model checker.

6.2 Example 2: A Transport Service Handler

This example was first introduced in [BB87]. It consists of a simple transport service handler (Figure 6.3). The service consists of three phases of conversation between two entities *A* and *B*:

- Connection establishment.
- Data transfer.
- Connection release.

6.2.1 Informal Description of the Service

The service is specified in terms of possible sequences of interactions (Transport Service Primitives) between the transport service provider and two session layer entities, at the transport service access points. The LOTOS specification describes only the interactions with respect to the service primitives. The parameters exchanged during the interactions are not considered.

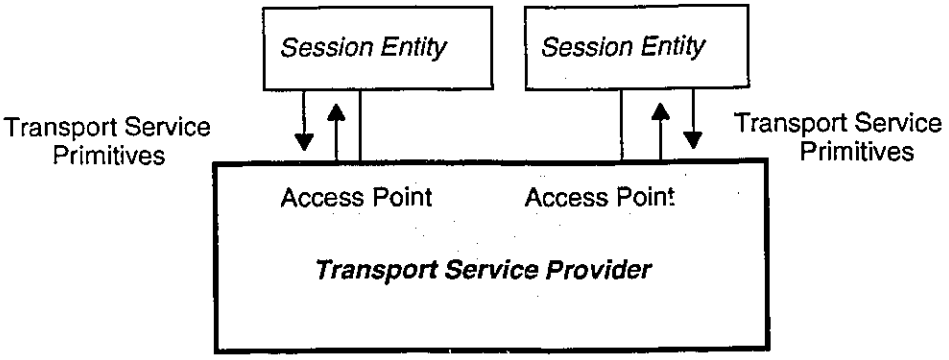


Figure 6.3 Transport Service

The following table describes the service primitives considered and their significance:

Primitives	Significance
ConReq	Connection Request
ConInd	Connection Indication
ConRes	Connection Response
ConCnf	Connection Confirmation
DisReq	Disconnection Request
DisInd	Disconnection Indication
DatReq	Data Request
DatInd	Data Indication

The service primitive sequences are expressed by time sequence diagrams (Figure 6.4):

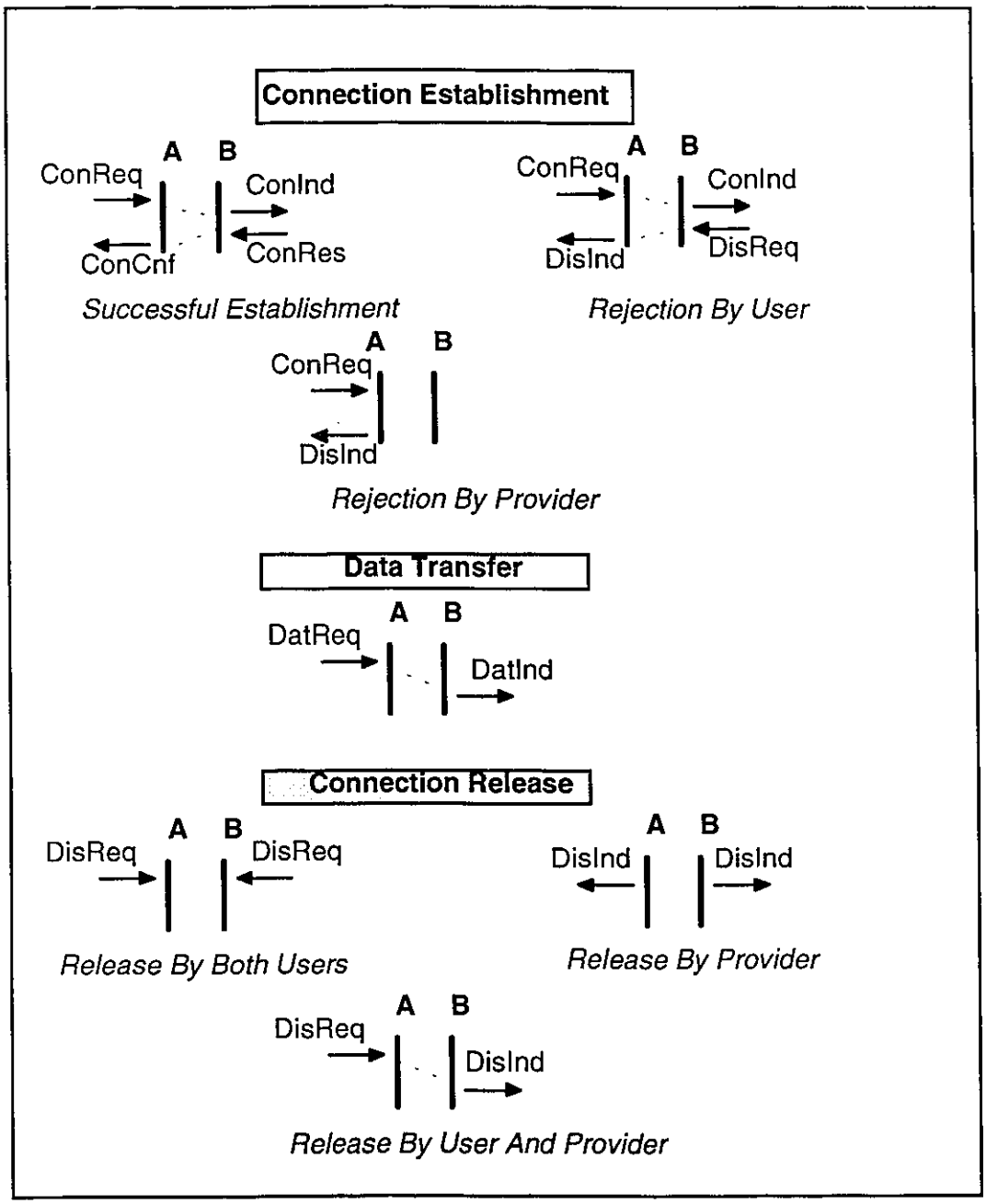


Figure 6.4. Transport Service Primitives

6.2.2 The LOTOS Specification

As in the previous example, we give here the main behaviour of the service; the complete specification as well as the symbolic behaviour tree are provided in appendix A. The main behaviour of the specification consists of a call to process *Handler* which is composed mainly of three processes that correspond to the three phases of the service. The following table gives a description of each of these processes along with their corresponding primitives.

Process	Phase	Primitives involved
Connection-phase	Connection establishment	*ConReq, ConInd, ConRes, ConCnf, DisReq, DisInd*
Data_phase	Data transfer	*DatReq, DatInd*
Termination_phase	Connection release	*DisReq, DisInd*

The main behaviour of the specification can then be written as a call to process *Handler*:

Handler[*ConReq,ConInd,ConRes,ConCnf, DatReq,DatInd,DisReq,DisInd*]

where process *Handler* is defined as follows:

Connection_phase[*ConReq,ConInd,ConRes,ConCnf,DisReq,DisInd*]
>>
(*Data_phase*[*DatReq,DatInd*]
[>
Termination_phase[*DisReq,DisInd*])
>>
Handler[*ConReq,ConInd,ConRes,ConCnf,DatReq,DatInd,DisReq,DisInd*]

Data_phase is only enabled when process *Connection_phase* terminates successfully. On the other hand, process *Termination_phase* can disrupt process *Data_phase* at any time before it terminates. Once process *Termination_phase* exits, it enables recursively the whole service again by calling process *Handler*.

6.2.3 The Symbolic Tree

Because of the fact that the specification is in "basic LOTOS", the tree generated here does not contain symbolic values. In other words, the exchange of parameters between the entities is not specified. Each state contains only a service primitive or an internal action.

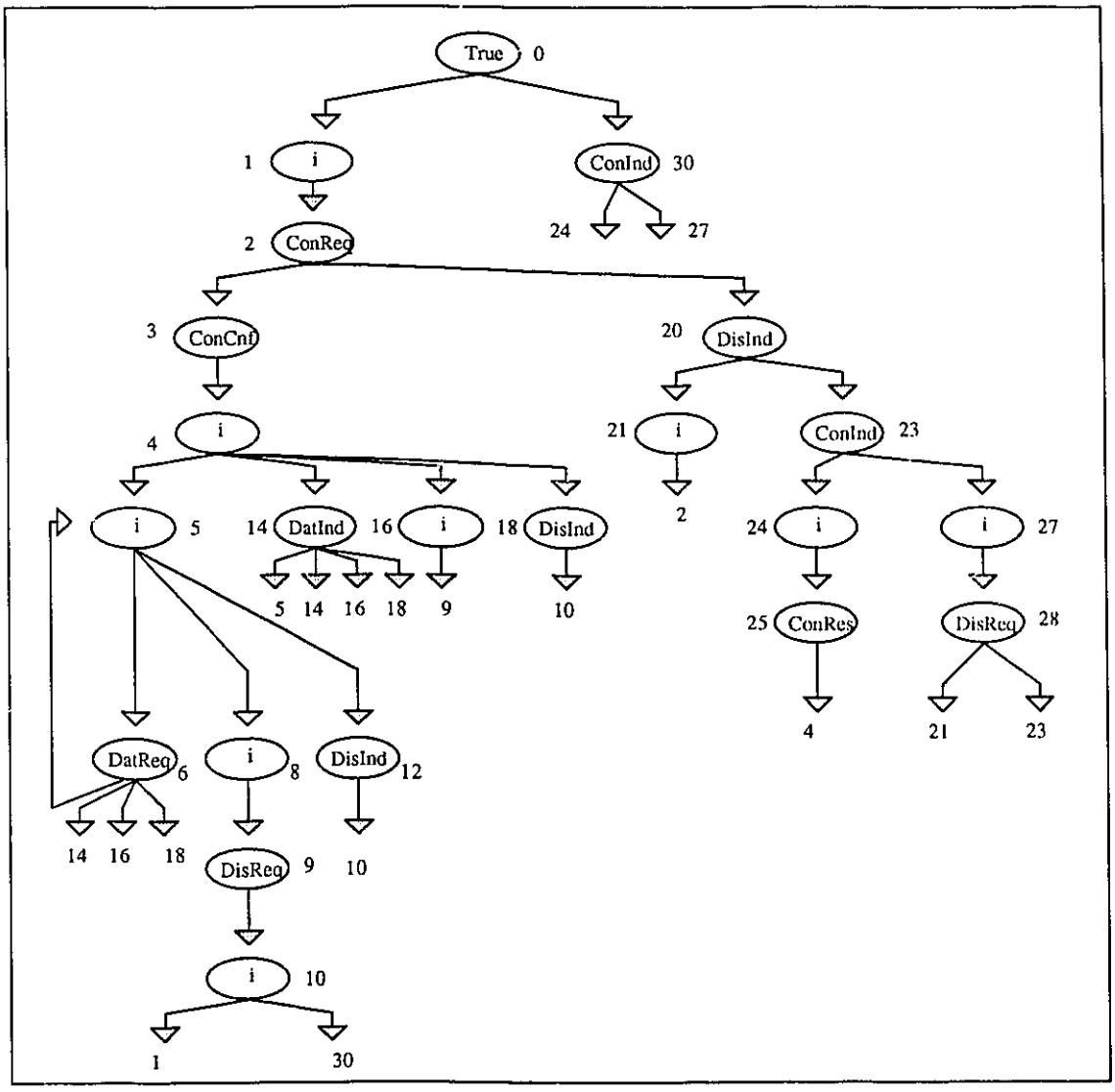


Figure 6. 5 Symbolic tree for the Transport Service Handler

6.2.4 Verification Phase

Starting from the informal requirements, many interesting properties can be derived. We give here some examples of such formulas that can be evaluated by LMC. It is to be noticed that value offers are not expressed in the formulas, simply because the specification does not consider the parameter exchanged during the interactions between the two session entities.

1) Absence of deadlock.

$$AG(\neg deadlock)$$

2) "Every request for connection *ConReq* can be granted by receiving a connection confirmation *ConCnf* or rejected, in which case, the user receives a disconnection indication *DisInd*."

$$AG(ConReq \Rightarrow AF(ConCnf \vee DisInd))$$

3) "If a connection indication *ConInd* is received, the entity should be able to respond *ConRes*, or reject it by requesting a disconnection *DisReq*."

$$AG(ConInd \Rightarrow AF(ConRes \vee DisReq))$$

4) "If a connection indication *ConInd* is received , the entity should always respond by sending a *ConRes*."

$$AG(ConInd \Rightarrow AF(ConRes))$$

This property obviously violates the service requirements. An entity should be able to reject a connection request. This property is evaluated to false by our model checker.

5) "The user can send a connection request *ConReq* and then, he can possibly request data *DatReq* without receiving a confirmation *ConCnf* to his request *ConReq*."

$$AG(ConReq \Rightarrow E[\neg ConCnf \upsilon DatReq])$$

This property mysteriously seems to violate the requirements of the service. However, it is satisfied. The property intends to verify whether it is possible for a user to send a *DatReq* without receiving a *ConCnf* to his request for connection *ConReq*. In fact, after sending a connection request, the user may receive a disconnection indication and then, another user may wish to establish connection with him. In that case, he can forward a connection response *ConRes* and then once the connection is established, he can send data requests. This is represented in the tree in the path 2, 20, 23, 24, 25, 5, 6. It can be seen easily that state 2 represents a *ConReq* and state 6 is labelled with *DatReq* and none of the intermediate states in the specified path represents a *ConCnf*.

Chapter 7

Conclusions and Future Work

We presented in this thesis a model checker for full LOTOS (LMC). The development of LMC is an enrichment to the University of Ottawa LOTOS tools. LMC can verify if certain properties expressed in temporal logic are true on the symbolic tree generated by SELA. The novel features of LMC are:

- A modified satisfaction relation, in order to deal with finite computations (section 4.4).
- The use of symbolic values in order to denote values to be obtained from the environment (section 5.4.1).
- The possibility of using "don't care" values for experiments (section 5.4.1).

The model checker can be used whenever a finite symbolic behaviour tree can be obtained from the given specification. Incomplete model checking on partial behaviour trees is also allowed in our system (section 4.5). This feature becomes interesting when the model is large as one may luckily detect errors on partial trees before having to generate the complete tree.

We have described in chapter 5 the set of properties expressible in LMC, and defined an extra predicate expressing deadlock in a LOTOS specification and its corresponding satisfaction relation (section 5.5). By doing so, we were able to express easily properties

related to the absence of deadlock on particular states, paths, or on the whole model representing the LOTOS specification.

We have attempted to combine the CTL operators and the LOTOS syntax for value offers when formulating the properties in order to make it easy to express properties in terms of LOTOS and hence, stay very close to the original specification. This feature simplified enormously the expression and understanding of the formulas. In chapter 6, we have shown by using examples, how some interesting properties can be derived, starting from the initial requirements. In particular, the use of "don't care" values (section 5.4.1) appears to be a novel feature of our tool.

The other existing model checker for full LOTOS is the system CÆSAR/CLÉOPÂTRE [FGMRS91] developed at the University of Grenoble and discussed in chapter 1. All possible values are represented explicitly in the models computed in this system while in our case, we use symbolic values (section 2.4.5). Model checking is meant to be used when it is possible to represent all the values explicitly in the model. However, for practical examples, values can be infinite and therefore it is impossible to represent them in a finite model. For this reason, in the CÆSAR/CLÉOPÂTRE system, the user has to define the range of each of the values to be provided by the environment by writing extra C code. This problem was avoided in our case, by using symbolic values but at the cost of including some unfeasible paths (section 5.4.3).

As discussed in section 1.3, the existence of model checking tools suggests a disciplined methodology for developing data communications software. We start with the user requirements. The requirements result in a specification which is expressed in LOTOS. The SELA tool obtains the model from the specification. The initial requirements are expressed in temporal logic, and the model checker is used to determine whether they are true or false. If they are false, a diagnostic system (not yet implemented, see below) can be used to find the reasons, after which the specification is revised in order to fix the errors.

The main drawbacks of our model checker can be summarized by he following points:

(i) As mentioned above, the possible existence of unfeasible paths could not be avoided due to the use of symbolic values in our model. The model checker is unable to detect contradictory conditions that contain symbolic values. For this reason, we suppose that these conditions are true. One way of solving this problem is to apply narrowing [RKKKL85] to selection predicates containing symbolic values. Another way is to compose the specification in parallel with a process that provides all the necessary values, or insert these directly in the specification.

(ii) A major problem that is encountered concerning the expression of properties related to "fairness" (see section 6.1.4). Since the behaviour of a system is generally described in a non-deterministic manner and interleaving semantics is used for parallelism, the model contains inevitably "unfair" execution sequences. These sequences can be considered as unrealistic behaviours [GRRV89]. For example, when dealing with network protocols where processes communicate over an imperfect or (lossy) channel, we may wish to disregard the unfair computation sequences; in this case the unfair computations are those in which a sender process continuously transmits messages without ever reaching the receiver due to erratic behaviour of the channel. Unfortunately, fair executions cannot be expressed by CTL. A new logic called CTL^F was introduced in [CES86] which has exactly the same semantics as CTL, except that all path quantifiers range over fair paths. In this thesis, we considered only CTL since it has sufficient expressive power to capture interesting properties of services, and fairness is not necessarily a desirable assumption.

(iii) The expression of some properties assumes that all messages exchanged between processes are distinguishable, which requires an infinite set of message identifications if infinite execution sequences are observed [GRRV89]. CTL is based on propositional logic, which means we can only quantify over paths but not over values (or propositions). Fortunately in our case, symbolic values replace actual values in the model which allows us to express properties over symbols instead of writing a formula for each value. However, when a message has a finite

number of values (for example modulo two as in the case of the alternating bit protocol for the sequence number), then we can of course derive a property for each value explicitly.

The previous discussion suggests several research directions for future work. These directions can be summarized by the following points:

(i) Effective use of a model checker presupposes the existence of a diagnostic system, capable of giving the user useful information on why some properties were not satisfied. We are investigating new ways to provide automated diagnostics without the intervention of the user, based on sequences extracted from the model (section 5.6). Previous work has been done on this subject and can be found in [RAS90].

(ii) Validation methods based on the generation of the complete reachability trees have their limitations, mainly due to the limitations in memory size. The major problem encountered by using model checking techniques is the explosion of the size of the state graph. Therefore, to be able to deal with real life protocols, intelligent techniques to compress and reduce the size of the graph will have to be developed. Also, optimized methods of generating and storing large behaviour trees will have to be used in order to extend the applicability of the tool to larger and more practical examples.

(iii) A promising new method recently developed in our LOTOS research group but not yet fully implemented, is the Goal Oriented Execution [HHLS92]. This method intends to find execution paths with certain characteristics, for example, execution paths that start with one particular action and ends with another one. Goal Oriented execution is able to find paths without having to generate the complete tree of the specification. The application of this method to model checking deserves to be studied in depth.

(iv) As mentioned above, fairness is a very important concept related to specification languages that are based on concurrent and non-deterministic computation models. In the context of specification languages, a variety of fairness properties have been proposed. For example, three kinds of fairness, namely process fairness, guard

fairness and channel fairness, are defined for CSP [Kuip83]. So far, little work has been done about fairness in LOTOS. A formal introduction of fairness in LOTOS was given in [WB90]; it is based on the standard semantics of the language together with a formalism that states restrictions on fair infinite execution sequences. Future work will have to deal with the formalization of the LOTOS computation model and its related fairness properties in the same formalism. Also, the logic CTL^F mentioned above could be implemented as an option in our tool.

Bibliography

[Ash92] P. Ashkar. Symbolic Execution of LOTOS Specifications. Master's Thesis, University of Ottawa, 1992.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The Temporal Logic of Branching Time. In *Proc. 8th Ann. Symp. on Principles of Programming Languages* (1981) 164-176; Journal version, *Acta Informatica*, (1983) 20:207-226.

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

[BBFM82] H.K. Berg, W.E Boebert, W.R. Franta, and T.G. Mober. *Formal Methods of Program Verification and Specification*.Prentice-Hall, Engelwood-Cliffs, NJ 1982.

[BC89] T. Bolognesi, M. Caneve. Squiggles: A tool for the Analysis of LOTOS Specifications. In K. Turner (ed.), *Formal Description Techniques*,(North-Holland, Amsterdam, 1989) 201-216.

[Bou91] R. Boumezbeur. Design, Specification, and Validation of Telephony Systems in LOTOS. Master's Thesis, University of Ottawa, 1991.

[CH87] Ana R. Cavalli, F. Horn. Proof of Specification properties by Using Finite State Machines and Temporal logic. In H. Rudin and C. West, (eds.), *Protocol Specification, Testing, and Verification, VII*, (North-Holland), 1987.

[CCITT87] Recommendation Z.100: Specification and Description Language SCL, CCITT SG X, Contribution Com X-R 15-E, 1987.

[CE81] E.M. Clarke, E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proc. Workshop on Logics of Programs, Lecture Notes in Computer Science*, (Springer, Berlin, 1981) 131:52-71.

[CES83] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications - A practical approach. In *Tenth Annual ACM Symposium on Principles of Programming Languages,* (Austin, Texas, January 1983) 117-126.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM TOPLAS*, 8(2):244-263, April 1986.

[CP88] P. Camurati, P. Prinetto. Formal Verification of Hardware Correctness: Introduction and Survey of current Research. *IEEE Computer*, 21(7):8-19, July 1988.

[EC81] E.A. Emerson, E.M. Clarke, Characterizing Correctness Properties of Parallel Programs as Fixpoints. In *Proc. 7th Internat. Coll. on Automata, Languages, and Programming, Lecture Notes in Computer Science*, (Springer, Berlin, 1981) 85:169-181.

[Emer90] E. Allen Emerson. Temporal and Modal Logic. In J. van Leewen, (ed.), *Handbook of Theoretical Computer Science.* (Elsevier Science Publishers B.V, 1990) Chapter 16.

[EM85] B. Ehrig, B. Mahr. *Fundamentals of Algebraic Specifications.* Springer-Verlag, 1985.

[Feh87] M.C. Fehri. A System for Validating and Executing LOTOS Data Abstractions (SVELDA). Master's Thesis, University of Ottawa, 1992.

[Fer89] J. C. Fernandez. Aldebaran: A Tool for Verification of Communicating Systems. Technical report SPECTRE, Laboratoire de Genie Informatique, Institut IMAG, 1989.

[Fer90] J. C. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 1990 13: 219-236.

[FGLR91] A. Fantechi, S. Gnesi, M. Leporatti, and G. Ristori. Model Checking For LOTOS. *3rd Workshop on Computer-Aided Verification*, 1991. To appear in *Lecture Notes in Computer Science*.

[FGMRS91] J.C. Fernandez, H. Graravel, L. Mounier, A. Rasse, C. Rodriguez, et J. Sifakis. Une boîte à outils pour la verification de programmes LOTOS. In O. Rafiq (ed.), *CFTP'91, Ingenierie des protocoles*, (Rennes, 1991) 479-500.

[Flo67] R.W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 1967 19:19:32.

[Fran86] N. Francez. *Fairness*. Springer-Verlag, New York, 1986.

[Gar89a] H. Garavel. Compilation et verification de programmes LOTOS. These de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[GHL88] R. Guillemot, M. Haj-Hussein, and L.Logrippo. Executing Large LOTOS Specifications. In S. Aggarwal and K. Sabnani, (eds.), *Protocol Specification, Testing, and Verification VIII*, (North-Holland, 1988) 399-410.

[GL89] R. Guillemot and L.Logrippo. Derivation of Useful Execution traces from LOTOS Specifications by using an Interpreter. In K.J. Turner, (ed.), *Formal Description Techniques*, (North-Holland, 1989) 311-325.

[GS90] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R.L. Probert, and H. Ural, (eds.), *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification* , (North-Holland, June 1990) 359-376.

[HC77] G. E. Hughes and M.J. Creswell. *An Introduction to Modal Logic.* Methuen and Co., 1977.

[HH89] M. Haj-Hussein. ISLA: An Interactive System for LOTOS Applications.Master's Thesis, University of Ottawa, 1989.

[HHLS92] M. Haj-Hussein, L. Logrippo, J. Sincennes. Goal-Oriented Execution for LOTOS. Technical Report, Computer Science Department, University of Ottawa, January 1992. To appear in *FORTE'92.*

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programs. In *Communications of the ACM*, (October 1969) 12(10):576-583.

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

[ISO8807] ISO, IS 8807. *Information Processing Systems - Open Systems Interconnection* - LOTOS : A formal Description Technique based on the Temporal Ordering of Observational behaviour, May 1989.

[ISO89] ISO, IS 9074. Estelle-A Formal Description Technique Based on an Extended State Transition Model, 1989.

[Jaou92] R. Jaouani. LOTOS Based Conformance Testing: the Theory and a Tool. Master's Thesis. University of Ottawa, 1992.

[JKP89] B. Jonsson, A. H. Khan, and J. Parrow. Implementing a Model Checking Algorithm by Adapting Existing Automated Tools. *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. Lecture Notes in Computer Science, (1989) 407:179-188.

[JT79] R.W. Jensen, C. C. Tonies. *Software Engineering*. Prentice-Hall, Engelwood, NJ. 1979.

[Karj92] G. Karjoth, C. Binding, and J.Gustafson. LOEWE: A LOTOS Engineering Workbench. To appear in Computer Networks and ISDN Systems.

[Lamp80] L. Lamport. Sometimes is Sometimes "Not Never"- on the Temporal Logic of Programs. In *Proc. 7th Annu. ACM Symp. on Principles of Programming Languages*, (1980) 174-185.

[LFH92] L. Logrippo, M. Faci, M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems* 23(5) 1992 325-342.

[LL91] E. Lallemand, Guy Leduc. On LOTOS tools and their usefulness to treat large specifications. OSI 95/ULg/A/11/TR/R/V2, 1991.

[Mil80] R. Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, (Springer-Verlag, 1980) No. 92.

[MP81b] Z. Manna, A. Pnueli. Verification of Concurrent Programs: the Temporal Framework. *Academic Press*, 215-273, 1981.

[OL82] S.S. Owicki, L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Trans. on Programming Languages and Systems*, 4(3) (1982) 455-495.

[PL91] S. Pavón, M. Llsmas. The Testing Functionalities of LOLA. In J. Quemada, J. Mañas, E. Vázquez, (eds.), *Formal Description Techniques III*, (North-Holland, 1991) 559-562.

[Pnue77] A. Pnueli. The temporal Semantics of Concurrent Programs. In *18th Symposium on Foundations of Computer Science*, 1977.

[Pnue81] A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, (1981) 13:45-60.

[QFP 88] J. Quemada, S. Pavón, and A. Fernández. Transforming LOTOS Specifications with LOLA: The Parametrized Expansion.In K. J. Turner, (ed.), *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88*, (North-Holland, Amsterdam, September 1988) 45-54.

[QS83] J. P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems - A Temporal Logic to Deal with Fairness. *Acta Informatica*,19:195-220, 1983.

[Ras90] A. Rasse. CLEO: Diagnostic des Erreurs en XESAR. These de Doctorat, Institut National Polytechnique de Grenoble, June 1990.

[RKKL85] P. Rety, C. Kirchner, H. Kirchner, and P. Lescanne. Narrower: A new Algorithm for Unification and its Application to Logic Programming. In Dijon, (ed.), *Proceedings of the First International Conference on Rewriting Techniques and Applications*, 141-157, 1985.

[Rod88] C. Rodriguez. Specification et Validation de Systemes en XESAR. These de Doctorat, Institut National Polytechnique de Grenoble, May 1988.

[Sif86] Joseph Sifakis. A Response to Amir Pnueli's "Specification and Development of Reactive Systems". In IFIP, (Dublin, Ireland, 1986) 1183-1187.

[vE89] P. van Eijk. The Design of a simulator tool. In P. van Eijk et al., (eds.), *The Formal Description Technique LOTOS*. (North-Holland, Amsterdam, 1989) 351-390.

[WB90] C. Wu, G. Bochmann. Fairness in LOTOS. Technical report, Université de Montreal, Publication # 769, 1990.

[Wes91] C. H. West. Protocol Validation Principles and Applications. Technical Report, 1991.

Appendix A: Two LOTOS Specifications

A.1 LOTOS Specification of the Datalink Service Provider

```
specification datalink[ get, give, send, receive ]:noexit

library  Boolean endlib                    (* Abstract Data Types Part *)

type sequeceNumber is Boolean              (* Type sequence number *)
       sorts seqNum
       opns  0       :                     -> seqNum
             inc     : seqNum              -> seqNum
             equal   : seqNum,seqNum       -> Bool

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

type bitString is Boolean      (* Type of messages is a string of bits *)

        sorts bitString
        opns  empty  :                                -> bitString
              equal  : bitString,bitString            -> Bool

        eqns  ofsort Bool forall x : bitString
                          equal(x,x) = true
endtype
```

```
type Frame is Boolean (* Messages exchanged are acknowledgments (ack) *)
        sorts Frame                  (* or  data (info) *)
        opns  info,ack:                    -> Frame
              equal   : Frame,Frame        -> Bool
        eqns   ofsort Bool forall x : Frame
                           equal(x,x) = true;
                           equal(ack,info) = false;
                           equal(info,ack) = false
endtype
            (* Control Part *)
behaviour

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

where

 process transmitter [get,tout,send,receive] (seq:seqNum) :noexit :=

    get ?data:bitString
   ;sending [tout,send,receive] (seq,data)
    >> transmitter [get,tout,send,receive] (inc(seq))

     where

    process sending[tout,send,receive](seq:seqNum,data:bitString:exit :=
            send !info !seq !data
            ;  ( receive !ack !inc(seq) !empty ; exit
                 []
                 tout
               ; sending [tout,send,receive] (seq,data)

              )
      endproc
endproc

process receiver [give,send,receive] (exp:seqNum) : noexit :=

    receive !info ?rec:seqNum ?data:bitString
   ; ( [rec = exp ] ->
          give !data
         ;send !ack !inc(rec) !empty
         ; receiver [give,send,receive] (inc(exp))

    []  [inc(rec) = exp] ->
          send !ack !inc(rec) !empty
         ; receiver[give,send,receive] (exp)
      )
 endproc
```

```
process line [tout,send,receive] : noexit  :=

     send ?f:Frame ?seq:seqNum ?data:bitString
    ; (  receive !f !seq !data
         ;line [tout,send,receive]
       []
         i
         ; tout
         ; line[tout,send,receive]
      )
endproc

endspec
```

A.2 LOTOS Specification of the Transport Service Handler

```
(* TS-Handler *)

 specification Handler[ConReq,ConInd,ConRes,ConCnf,
                       DatReq,DatInd,DisReq,DisInd]:noexit

 behaviour

    Handler[ConReq,ConInd,ConRes,ConCnf,
          DatReq,DatInd,DisReq,DisInd]

(*-------------------------------------------------------------*)

  where process Handler[ConReq,ConInd,ConRes,ConCnf,
                        DatReq,DatInd,DisReq,DisInd] :noexit:=

        Connection_phase[ConReq,ConInd,ConRes,ConCnf,DisReq,DisInd]
       >>
        ( Data_phase[DatReq,DatInd]
         [>
          Termination_phase[DisReq,DisInd] )
       >>
        Handler[ConReq,ConInd,ConRes,ConCnf,
              DatReq,DatInd,DisReq,DisInd]

(*-------------------------------------------------------------*)

     where process Connection_phase[CRq,CI,CR,CC,DR,DI] :exit:=

             i ; Calling[CRq,CI,CR,CC,DR,DI]
             []
             Called[CRq,CI,CR,CC,DR,DI]


            where process Calling[CRq,CI,CR,CC,DR,DI] :exit:=
                   CRq ;
              (CC ; exit
                   []
                   DI ; Connection_phase[CRq,CI,CR,CC,DR,DI])
                  endproc
                  process Called[CRq,CI,CR,CC,DR,DI] :exit:=
                  CI ;
                    (i ; CR ;exit
                  []
                    i ; DR ;Connection_phase[CRq,CI,CR,CC,DR,DI])
                  endproc

            endproc

(*-------------------------------------------------------------*)
```

```
        process Data_phase[DtR,DtI]:noexit:=
                (i ; DtR ;Data_phase[DtR,DtI]
               []
                DtI ; Data_phase[DtR,DtI])
               endproc

(*-------------------------------------------------------------*)

        process Termination_phase[DR,DI]:exit:=
                (i ;DR ;exit
               []
                DI ;exit)
                endproc

    endproc (* handler *)

endspec
```

A.3 Symbolic Trees Generated by SELA

A.3.1 Datalink Service Provider

```
bh0 * 1 get ?bitString@1:bitString [55]
bh1 * | 1 send !info !0 !bitString@1 [62,89]
bh2 * | | 1 i (specified explicitly) [93]
bh3 * | | | 1 i (hiding: tout) [65,94]
bh4 * | | | | 1 send !info !0 !bitString@1 [62,89] ==> again bh2
    * | | | 2 receive !ack !inc(0) !empty
                ([ack=info] and [inc(0)=0] and [empty=bitString@1] ) [63,90]
bh5 * | | | 1 i (enable: exit) [63]
bh6 * | | | | 1 get ?bitString@2:bitString [55] ==> again bh1
    * | | | 3 receive !info !0 !bitString@1 [74,90]
bh7 * | | | | 1 [0=0] give !bitString@1 [76]
bh8 * | | | | | 1 send !ack !inc(0) !empty [77,89]
bh9 * | | | | | | 1 i (specified explicitly) [93]
bh10* | | | | | | | 1 i (hiding: tout) [65,94]
bh11* | | | | | | | | 1 send !info !0 !bitString@1 [62,89] ==> again bh9
    * | | | | | | 2 receive !ack !inc(0) !empty [63,90]
bh12* | | | | | | | 1 i (enable: exit) [63]
bh13* | | | | | | | | 1 get ?bitString@3:bitString [55]
bh14* | | | | | | | | | 1 send !info !inc(0) !bitString@3 [62,89] ==> again bh9
    * | | | | | | 3 receive !info !inc(0) !empty
                      [info=ack]  [74,90] ==> again bh7
    * | | | | 2 [inc(0)=0] send !ack !inc(0) !empty [81,89]
bh15* | | | | | 1 i (specified explicitly) [93]
bh16* | | | | | | 1 i (hiding: tout) [65,94]
bh17* | | | | | | | 1 send !info !0 !bitString@1 [62,89] ==> again bh15
    * | | | | | 2 receive !ack !inc(0) !empty [63,90]
bh18* | | | | | | 1 i (enable: exit) [63]
bh19* | | | | | | | 1 get ?bitString@3:bitString [55]
bh20* | | | | | | | | 1 send !info !inc(0) !bitString@3 [62,89] ==> again bh15
    * | | | | 3 receive !info !inc(0) !empty
                  [info=ack]  [74,90] ==> again bh7
```

A.3.2 Transport Service Handler

```
bh0  * 1 i (specified explicitly) [29]
bh1  * | 1 ConReq [35]
bh2  * | | 1 ConCnf [36]
bh3  * | | | 1 i (enable: exit) [36]
bh4  * | | | | 1 i (specified explicitly) [52]
bh5  * | | | | | 1 DatReq [52] ==> again bh4
     * | | | | | 2 i (specified explicitly) [60]
bh6  * | | | | | | 1 DisReq [60]
bh7  * | | | | | | | 1 i (enable: exit) [60] ==> again bh0
     * | | | | | | 3 DisInd [62] ==> again bh7
     * | | | | | 2 DatInd [54] ==> again bh4
     * | | | | | 3 i (specified explicitly) [60] ==> again bh6
     * | | | | | 4 DisInd [62] ==> again bh7
     * | | | 2 DisInd [38]
bh8  * | | | 1 i (specified explicitly) [29] ==> again bh1
     * | | | 2 ConInd [42]
bh9  * | | | | 1 i (specified explicitly) [43]
bh10 * | | | | | 1 ConRes [43] ==> again bh3
     * | | | | | 2 i (specified explicitly) [45]
bh11 * | | | | | 1 DisReq [45] ==> again bh8
     * 2 ConInd [42] ==> again bh9
```

Appendix B: Examples using LMC

The following table shows the temporal logic operators used by LMC and their corresponding operators used for the implementation:

Original Operators	Implementaion Operators
AG	ag
EG	eg
AX	ax
EX	ex
AF	af
EF	ef
A[f_1 υ f_2]	all[f_1 until f_2]
E[f_1 υ f_2]	some[f_1 until f_2]
⇒	->
∨	or
∧	&
¬	~

The following session demonstrates the use of LMC to evaluate the properties related to the case studies that were described in chapter 6.

```
                    LMC
                MODEL CHECKER
                 FOR LOTOS

/* Main menu of LMC */
-------------------------------------------------------------
-------------------------------------------------------------
<1> (l)    List LOTOS specifications '.l'
<2> (p)    List Prolog files  '.pl'
<3> (d)    List directory  '*.*'
<4> (v)    Verify  Temporal Logic Formulae
<5> (vi)   vi editor
<6> (pwd)  pwd (current directory)
<7> (u)    Other unix options
<8> (q)    Exit  LMC?
-------------------------------------------------------------
-------------------------------------------------------------


enter <command> or m[enu] to return to main:
==> v

% compiling file
/tmp_mnt/home/prga/usr7/grad/bghribi/newidea/brahim_tnt.pl
% brahim_tnt.pl compiled in module user, 1.334 sec 6,676 bytes


 Press (h) for help or <RETURN> to proceed:

 Enter the file name or <RETURN> to cancel
 ==> datalink
/* Name of the specification */

% compiling file
/tmp_mnt/home/prga/usr7/grad/bghribi/newidea/datalink.pl
% datalink.pl compiled in module user, 2.217 sec 9,932 bytes

 Enter the tree file name or <RETURN> to cancel
 ==> datalinkdata    /* Behaviour tree file generated by SELA */
% compiling file
/tmp_mnt/home/prga/usr7/grad/bghribi/newidea/datalinkdata.pl
% datalinkdata.pl compiled in module user, 1.634 sec 19,600 bytes


 Enter Formula    :

     > ag(~deadlock).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y
```

```
Enter Formula    :

     > ag( get!@1 -> ef(give!@1)).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :

     > ag(get!* -> ef(give!*)).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :

     > ag( receive!*!*!* -> ax(all(~ receive!*!*!* until send!*!*!* ))).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :

     > ag(send!ack!0!empty -> af(receive!info!inc(0)!* &
                 ax(send!ack!0!empty) or (receive!info!0!*))).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :
```

```
     > ag( send!info!*!* -> ax(all(~ send!info!*!* until
                                        receive!info!*!* ))).

     Checking Formula...

*********************************************************
    FORMULA  IS NOT SATISFIED
*********************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu:


                      LMC
                  MODEL CHECKER
                   FOR LOTOS

-------------------------------------------------------------
-------------------------------------------------------------
<1> (l)    List LOTOS specifications '.l'
<2> (p)    List Prolog files  '.pl'
<3> (d)    List directory  '*.*'
<4> (v)    Verify  Temporal Logic Formulae
<5> (vi)   vi editor
<6> (pwd)  pwd (current directory)
<7> (u)    Other unix options
<8> (q)    Exit  LMC?
-------------------------------------------------------------
-------------------------------------------------------------

enter <command> or m[enu] to return to main:
==> v

% compiling file
/tmp_mnt/home/prga/usr7/grad/bghribi/newidea/brahim_tnt.pl
% brahim_tnt.pl compiled in module user, 1.350 sec 1,532 bytes


 Press (h) for help or <RETURN> to proceed:

 Enter the file name or <RETURN> to cancel
 ==> ts_handler                      /* Transport service handler */

% compiling file
/tmp_mnt/home/prga/usr7/grad/bghribi/newidea/ts_handler.pl
% ts_handler.pl compiled in module user, 1.234 sec 1,820 bytes

 Enter the tree file name or <RETURN> to cancel
 ==> ts_handlerdata
            /* Behaviour tree  generated by SELA */

% compiling file
/tmp_mnt/home/prga/usr7/grad/bghribi/newidea/ts_handlerdata.pl
% ts_handlerdata.pl compiled in module user, 0.634 sec 7,324 bytes
```

```
Enter Formula    :

     > ag(~deadlock).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :

     > ag('ConReq' -> af('ConCnf' or 'DisInd') ).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :

     > ag('ConInd'  -> af('ConRes' or  'DisReq')).

     Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :

     > ag('ConInd' -> af('ConRes') ).

     Checking Formula...


************************************************************
    FORMULA  IS NOT SATISFIED
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu: y


 Enter Formula    :

     > ag('ConReq' -> some(~('ConCnf')  until  'DatReq' ) ).
```

```
    Checking Formula...

************************************************************
     FORMULA  HOLDS
************************************************************

   Press (y) for more Formulae or <RETURN> to  quit to Main menu:


                      LMC
                  MODEL CHECKER
                   FOR LOTOS

------------------------------------------------------------
------------------------------------------------------------
<1> (l)     List LOTOS specifications '.l'
<2> (p)     List Prolog files  '.pl'
<3> (d)     List directory  '*.*'
<4> (v)     Verify  Temporal Logic Formulae
<5> (vi)    vi editor
<6> (pwd)   pwd (current directory)
<7> (u)     Other unix options
<8> (q)     Exit  LMC?
------------------------------------------------------------
------------------------------------------------------------

enter <command> or m[enu] to return to main:
==> q
/* End of the session */
```

Appendix C: Diagnostics

This example illustrates the use of LMC and of ISLA for diagnostics. We consider the formula:

ag((send!info!*!*) -> ax(all(~(send!info!*!*) until (receive!info!*!*)))).

This formula was described in chapter 6 for example 1 (Datalink Service Provider). As expected, the formula is false because it states that whenever a message of type info is sent, then no message of type info is sent until the receiver receives a message of type info. However, we know that in the case of a timeout, the message of type info is retransmitted which allows the possibility of having a message of type info sent twice before any receive happens.

```
/* Using the Model Checker*/
---------------------------------

ENTER TEMPORAL LOGIC FORMULA   :

  >

   CHECKING FORMULA ...


**********************************************************
  FORMULA  IS NOT SATISFIED
**********************************************************
```

The file Traversal contains the list of states visited during the model checking procedure. It shows whether synchronization occured between the actions of the state and of the formula. For example, in this case, at state *1*, *send!info!*!** failed to synchronize with the label of the state. At state 2, synchronization was possible and we can see that following the sequence 2, 5, two successive (*send!info!*!**) were encountered and no intermediate *recieve* was detected. This explains why the formula did not hold.

```
/* File Traversal */
---------------------------

checking(failed,1,send!info!*!*).
checking(success,2,send!info!*!*).
checking(failed,5,receive!info!*!*).
checking(success,5,send!info!*!*).
```

The following is an execution of the specification using ISLA:

```
/* Using ISLA for Diagnostics */
---------------------------------
=================================================================
No Internal actions

 <1>- get ?data:bitString ---> bh1  [55]   /* The only possible action at the moment */

=================================================================
==> 1
 Enter a value for data:bitString
 => empty                                  /* We choose for example an empty string */

=================================================================
No Internal actions

 <1>- send !info:Frame !0:seqNum !empty:bitString ---> bh1  [62,89]  /* our empty string is sent */

=================================================================
==> 1
 Passed evaluated value ==> info                    /* The passed parameters */

 Passed evaluated value ==> 0

 Passed evaluated value ==> empty

=================================================================
No Internal actions

 <1>- i (specified explicitly) ---> bh1  [93]
 <2>- receive !info:Frame !0:seqNum !empty:bitString ---> bh2  [74,90]

=================================================================
==> 1
  Internal event is executed

=================================================================
No Internal actions
```

```
 <1>- i (hiding: tout) ---> bh1   [65,94]        /* Timeout occurs */

=================================================================
==> 1
  Internal event is executed

=================================================================
No Internal actions

 <1>- send !info:Frame !0:seqNum !empty:bitString ---> bh1   [62,89]

=================================================================
```

/* Another send is detected before a receive is encountered, therefore we reached our counter example */

Notice that at some point, there were two possible actions to choose from (indicated in bold). We avoided the *receive* action because we wanted to detect a path in which no *receive* is encountered after the first *send* until we meet a second *send*. It is obvious that the other action is the only alternative. By doing so, we were able to reach a state in which a new *send* is performed without an intermediate *receive*.