A Model Checker For LOTOS

By

Brahim Ghribi

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

UNIVERSITÉ D' OTTAWA **UNIVERSITY OF OTTAWA**

© Brahim Ghribi, Ottawa, Canada, 1992

Thesis Commitee

Supervisor:

Prof. Luigi Logrippo, University of Ottawa.

Examiners:

Prof. Gerald M. Karam, Carleton University.

Prof. Ali Mili, 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
1.1 Background 1
1.1.1 Interactive Simulation 2
1.1.2 Symbolic Execution 3
1.1.3 Formal Verification 3
1.2 The University of Ottawa LOTOS "toolkit" 5
1.3 Motivation of The Thesis 6
1.4 The Existing Model Checkers for LOTOS 7
1.5 Organization of the Thesis 8
Chapter 2
From LOTOS Specifications
to Behaviour Trees 9
2.1 Introduction 9
2.2 LOTOS Data Types 10
2.2.1 Signature 10
2.2.2 Equations 11
2.3 The Control Component 12
2.3.1 Inaction 14
2.3.2 Action prefix 14
2.3.3 Choice 14
2.3.4 Parallel composition 15
2.3.5 Hiding 16
2.3.6 Sequential composition (enabling) 16
2.3.7 Successful termination 17
2.3.8 Disabling 17
2.3.9 Guarded behaviour 18
2.3.10 Process instantiation 18
2.4 Symbolic Expansion of LOTOS Specifications 19
2.4.1 Explicit Stop or Deadlock 21
2.4.2 Duplicate Behaviours 22
2.4.3 Syntactically Equivalent Behaviours 23
2.4.4 Upper Bounds 24
2.4.5 Symbolic Values 25
2.5 Chapter Summary 27

Chapter 3
The Computation
Tree Logic CTL 28
3.1 Introduction 28
3.2 Definitions 29
3.2.1 Labelled Transition Systems (LTS) 29
3.2.2 Kripke Structures (KS) 30
3.2.3 Transformation from LTS to KS 31
3.3 Computation Tree Logic (CTL) 32
3.3.1 Definition (CTL) 33
3.3.2 Satisfaction Relation of CTL 33
3.3.3 Validity and Satisfiability of Formulas 36
3.4 Correctness Properties 37
3.4.1 Safety Properties 38
3.4.2 Liveness Properties 39
3.5 Chapter Summary 41
Chapter 4
The Model Checking
Algorithms 42
4.1 The Checking Algorithms 42
4.2.1 The Next Time Operators 43
4.2.2 The Global Operators 44
4.2.3 The Future Operators 47
4.2.4 The Until Operators 49
4.2 Complexity of The Algorithms 51
4.3 Adaptation to LOTOS 52
4.4 A Modified Satisfaction Relation 53
4.5 The Modified Algorithms 55
4.6 Chapter Summary 62
Chapter 5
LMC Design
and Implementation 63
5.1 General Structure of LMC 63
5.2 The Interface Module 65
5.3 The Transformer 66
5.3.1 The Tree Model 66
5.3.2 An Example 67
5.4 The Model Checker 69
5.4.1 Formulation of the Properties 70
5.4.2 The Synchronization Procedure 71
5.4.3 Evaluation of Selection Predicates and Guards 73
5.5 Properties Handled by LMC 77
5.6 Diagnostics 79

Chapter 6
Case Studies 82
6.1 Example 1: A Data Link Service Provider 82
6.1.1 Informal Description of the Service Provider 82
6.1.2 The LOTOS Specification 83
6.1.3 The Symbolic Tree 84
6.1.4 Verification Phase 87
6.2 Example 2: A Transport Service Handler 88
6.2.1 Informal Description of the Service 89
6.2.2 The LOTOS Specification 91
6.2.3 The Symbolic Tree 92
6.2.4 Verification Phase 93
Chapter 7
Conclusions
and Future Work 95

Bibliography 100
Appendix A: Two LOTOS Specifications 107
Appendix B: Examples using LMC 114
Appendix C: Diagnostics 120

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 method*s* and model-based methods.

Proof-based method*s* 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.

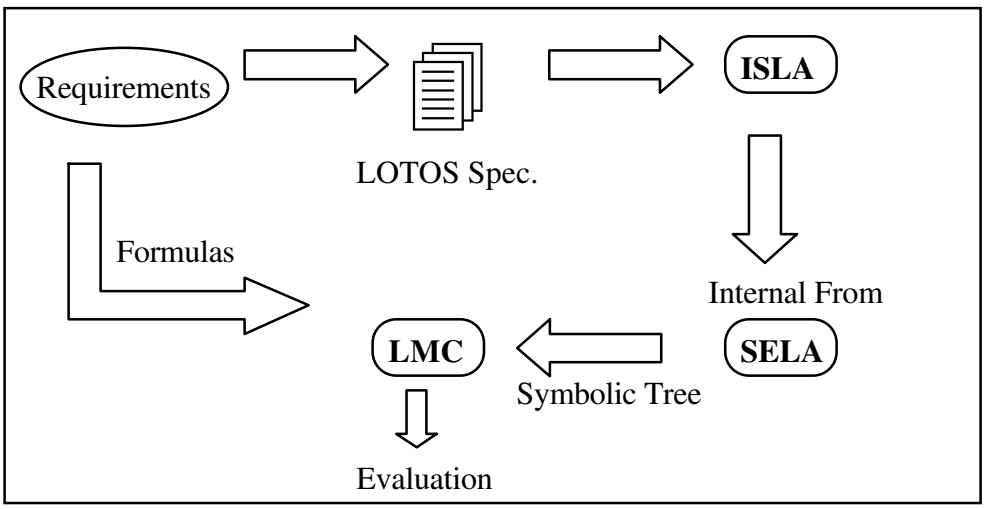


Figure 1.1 The Validation Environment

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) ∈ 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:

$B = B_1 \mid\mid\mid B_2 = B_1 \mid [ ] \mid B_2$ *(Interleaving)*
$B = B_1 \mid\mid B_2 = B_1 \mid [g_1,...,g_n] \mid B_2$ *(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*.

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

$E_1,..., E_n$ 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:

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

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

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 ...*

```
(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.

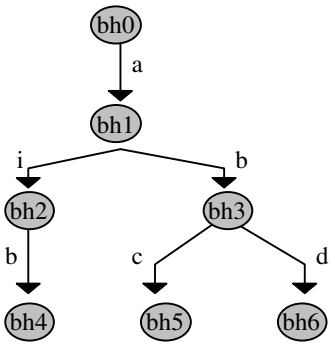
can be obtained. By iteratively applying the same rules on the resulting behaviour expressions, the whole behaviour tree can be generated (if it is finite).

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

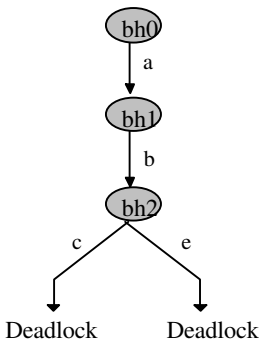
The exploration of a branch will end whenever a stop action in the original behaviour is executed, or a deadlock results from synchronization.

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

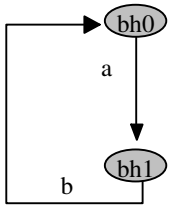
In the most simple case, repetition can be detected whenever a current behaviour is found to be *syntactically equal* to a previous one. Two behaviours are said to be syntactically equal, if they are identical as chains of characters.

EXAMPLE

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

The corresponding behaviour tree is:

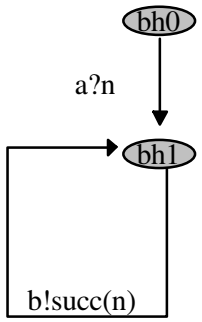
Note that the sequence of actions *a b* is repeated infinitely, and in fact *bh0* and *bh2* are duplicates. Once the duplicate behaviours are detected, the corresponding behaviour tree is:



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

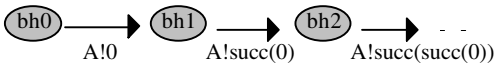
In some cases, behaviours are identical except for the value of some value expressions. This can be due to recursion, where the current behaviour is the instantiation of a process (or a set of processes in the case of parallelism) which had already been instantiated previously. The detection is achieved 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:



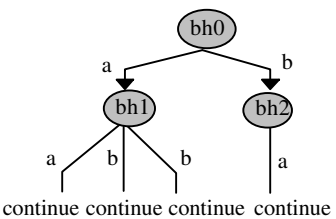
2.4.4 Upper Bounds

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:

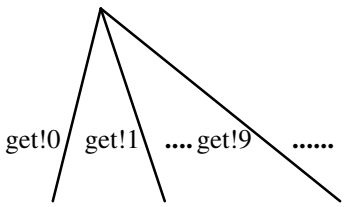
(i) The sort of the value identifier,
(ii) The symbol @ which expresses that the value is to be bound at a gate,
(iii) A numerical value that indicates the binding occurrence of the value.

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: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.

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

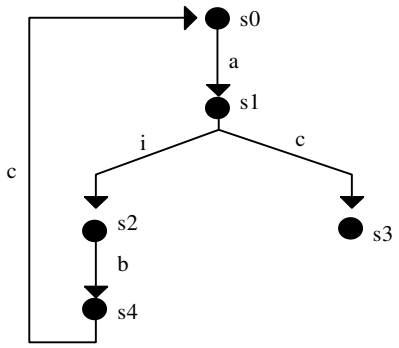
We let s_0, s_1, s_2, ... to range over S and a, b, c, ... to range over $\sum$.
A labelled transition in Δ is a triple $<s_1, a, s_2>$ usually represented by $s_1 \xrightarrow{a} s_2$.
A state $s \in S$ is called *terminal* if there is no transition from s in Δ.

Example

consider the following LOTOS behaviour:

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

The corresponding LTS is the following:



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

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 ,$\sum$,Δ> be a labelled transition system. We obtain the kripke structure
ks = <K, q_0, Prop, L, $\Re$> as follows:

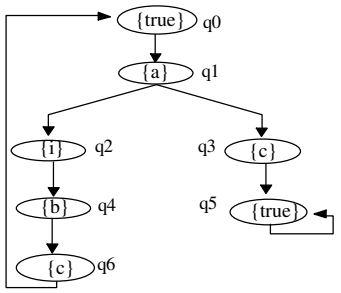
- The set Prop of atomic propositions is defined as $\sum$ (Prop = $\sum$).
- For every pair of transitions $s_1 \xrightarrow{a} s_2$, $s_2 \xrightarrow{b} s_3$ in the LTS, <$s_1 \xrightarrow{a} s_2$, $s_2 \xrightarrow{b} s_3$> is in $\Re$.
  For every transition $s_0 \xrightarrow{a} s$ in the LTS, the 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 $\Re$.

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

- 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:



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

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

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 $(q_0, q_1, q_2, ...)$ 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 $(q_0, q_1, q_2, ...)$ from q_0 in ks
$\forall i \geq 0 \quad q_i \models f$.

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

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

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

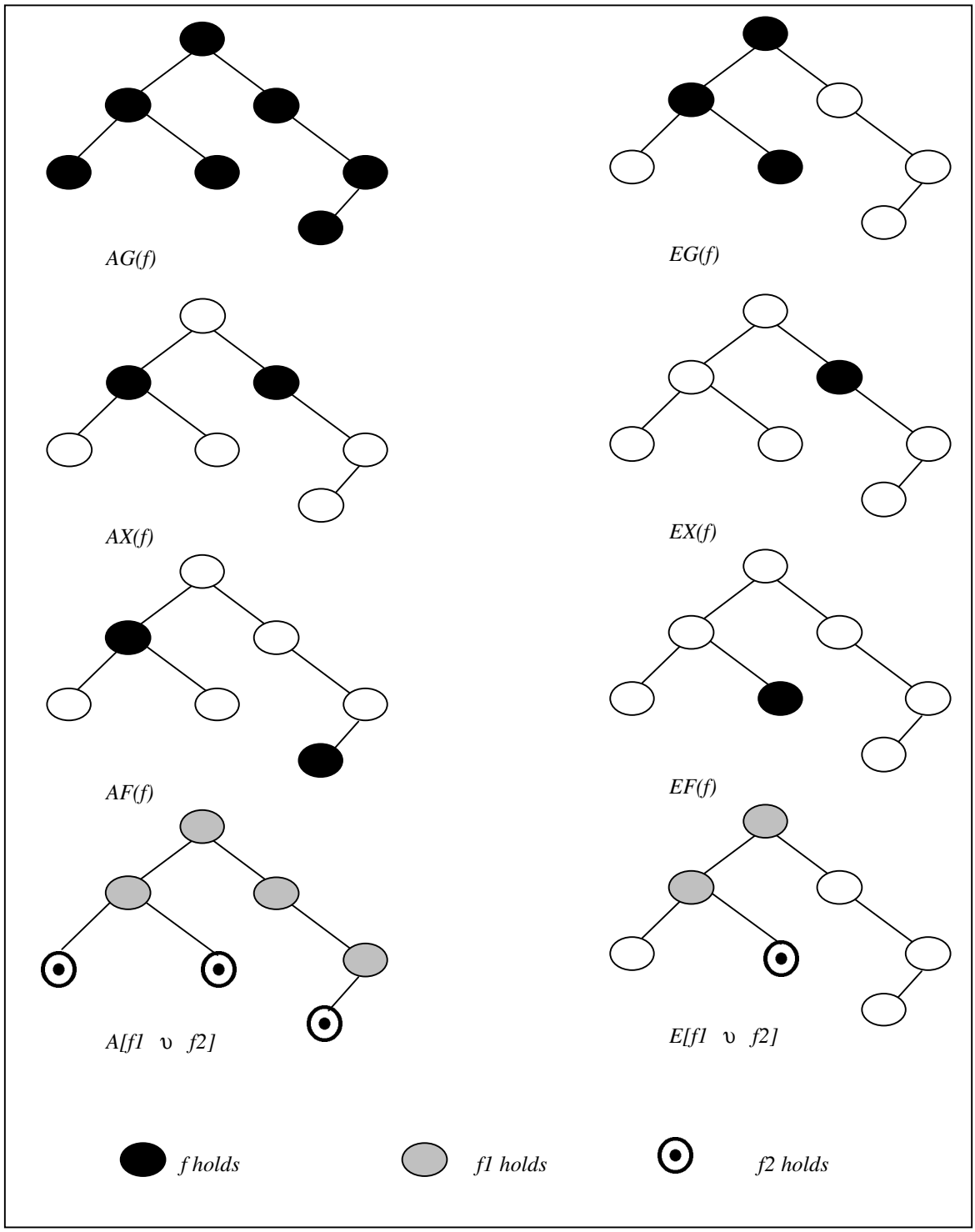


Figure 2.1 CTL Operators

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) $\mathrm{A}[f_1 \upsilon f_2] \wedge \neg(f_1 \wedge f_2)$ 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:

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

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

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

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:

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

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.