National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

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.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

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.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

LOTOS Based Conformance Testing

The theory and a tool

By

Rafik Jaouani

Thesis Submitted
to the School of Graduate Studies
in partial fulfilment 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

© Rafik Jaouani, Ottawa, Canada, 1992

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

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-85810-9

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

To my parents

and Anissa

Abstract

Computers in a network have to obey to well-defined protocols in order to communicate properly. These protocols can be very complex and their implementation is often subject to errors. One question arises after implementing a communication protocol: does the implementation conform to its specification? The process of answering this question is called *conformance testing*. The introduction of formal description techniques, in particular LOTOS, made it possible to formalize the problem and to develop formal methods to check the conformance of implementations to their specifications.

To test for conformance, tests can be derived from specifications and then applied to implementations. The *CO-OP method* is the basic test case generation method for LOTOS. It is used in order to generate canonical testers from specifications. The method that has been published in the literature can only deal with specifications that do not involve recursion (finite behaviours). We have generalized the CO-OP method in such a way as to remove this restriction: the only restriction we have is that the specification must have an expansion that can fit in memory. The generalized method also expands the original method to support a large subset of Full LOTOS behaviours.

In most cases, testing does not prove conformance but attempts to reduce the errors in implementations. We have developed an algorithm for proving conformance in the special case of Basic LOTOS behaviours. All algorithms presented in the thesis were implemented in what we call the LOTEST tool.

Acknowledgments

My special thanks go to my supervisor Dr. Luigi Logrippo for his support, advice and patience; without his directions, this thesis could have looked much different and less complete. I would like to thank my colleagues and friends at Ottawa University for their useful discussions and critiques; working among them made this work more fun and less stressful.

I would like to express my gratitude to the Tunisian Government and the University Mission of Tunisia for providing the financial support under the scholarship program sponsored by CIDA. As well, I should acknowledge the financial support of the Canadian Institute for Telecommunication Research.

My deep thanks go to my parents, for their patience and support throughout my studies; their advice and encouragement always helped me achieve my goals in life and it will always do.

Finally, I would like to thank Dr. Abdellatif Obaid and Dr. Robert L. Probert for accepting the responsibility of correcting this work.

Table Of Contents

Chapter 1

Introduction 1

1.1 Communication protocols 1

1.2 The OSI reference model 2

1.3 Formal Description Techniques 3

1.4 LOTOS 3

1.4.1 LOTOS data types 4

1.4.2 The control component 6

1.5 Conformance testing 12

1.5.1 Test derivation 13

1.5.2 Test selection 14

1.6 Motivation of the thesis 15

1.7 Summary of results 16

1.8 Organization of the thesis 17

Chapter 2

A Formal Approach

To Conformance Testing 19

2.1 Labelled transition systems 19

2.2 The implementation relation 22

2.3 Canonical testers 24

2.4 The CO-OP method 27
2.4.1 Compulsory and options sets 27
2.4.2 The expression *B after* μ 29
2.4.3 Initial behaviour of the tester 29
2.4.4 Building the canonical tester 30
2.4.5 Optimizing the canonical tester 32
2.4.6 Extracting test cases from the canonical tester 33
2.5 Chapter summary 36

Chapter 3
Generalization Of
The CO-OP Method 37
3.1 Critique of the current CO-OP method 37
3.1.1 Recursive behaviour expressions 37
3.1.2 Form of the canonical tester 41
3.2 State set of a labelled transition system 41
3.3 The generalized CO-OP method for Basic LOTOS 44
3.3.1 Goal of the new method 44
3.3.2 Compulsory and options sets for the generalized CO-OP method 45
3.3.3 Finding the compulsory and the options sets 46
3.3.4 Finding the orthogonal set 48
3.3.5 Canonical testers and state sets 48
3.3.6 Building the canonical tester 53
3.3.7 Optimizing the canonical tester 56
3.4 Chapter summary 58

Chapter 4
The Conformance
Relation And Refusal Sets 59

4.1 Refusal sets and failure trees 59
4.2 Canonical trace equivalent 61
4.2.1 Building the canonical trace equivalent 62
4.2.2 Trace inclusion 65
4.3 Complete failure trees 67
4.4 Proving conformance using complete failure trees 71
4.5 Limitations of the approach 73
4.6 Chapter summary 73

Chapter 5
Canonical Testers
Of Full LOTOS Specifications 74
5.1 Parameterized labelled transition systems 74
5.2 Applying the CO-OP method to pLTSs 76
5.2.1 Overlapping transitions 76
5.2.2 Deadlocking transitions 83
5.2.3 Internal events with predicates 86
5.3 Chapter summary 89

Chapter 6
Implementation
Issues 90
6.1 The Ottawa University LOTOS tool kit 90
6.2 Expansion of LOTOS behaviour expressions 94
6.3 Using SELA and ISLA to build a symbolic execution tree 94
6.4 Some implementation details 96

Chapter 7
Conclusions 101

Appendix A
A Tool For LOTOS
Testing Theory 103
A.1 A brief overview of LOTEST 103
A.2 Command reference 104
A.2.1 Syntax conventions 104
A.2.2 Online help with commands 105
A.2.3 LOTEST command summary 106
A.2.4 Format of the trees used by LOTEST 114
A.3 Limitations 115
A.3.1 Size of the symbolic execution tree 115
A.3.2 Non tail recursion 115
A.3.3 Full LOTOS 116

Appendix B
Examples 117
B.1 A working session with LOTEST 117
B.2 Checking interesting properties with LOTEST 119
B.2.1 The tester of the tester 119
B.2.2 Refusal sets 123
B.3 A Full LOTOS example 124

Bibliography 128

Chapter 1

Introduction

In order to establish the context of our work, in this chapter, we will introduce briefly some basic terms and concepts that relate to our subject. First, we will present the following topics:

- Communication protocols.
- The OSI reference model.
- Formal Description Techniques (FDTs).
- LOTOS, the Language Of Temporal Ordering Specifications.
- Conformance testing.

Then, we will present the motivation of our work and how it relates to those topics.

1.1 Communication protocols

Computers in a network have to obey well-defined protocols in order to communicate properly. A protocol is a set of rules and conventions between the communicating participants. These protocols can be very complex and their implementation is often subject to errors. To make their implementation more manageable they are likely to be

designed in layers where each layer concentrates on providing a particular functionality. Well-defined interactions are provided between layers.

1.2 The OSI reference model

The International Standards Organization (ISO) and the Consultative Committee for International Telephony and Telegraphy (CCITT) have adopted the open systems interconnection model (OSI) as a framework within which standards can be developed for services and protocols. The OSI model is not a specification but a guide to be followed when describing layers in a network. The OSI model has 7 layers (see Table 1.1).

	Layer
7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data link
1	Physical

Table 1.1 : The OSI model

Many computer networks today are described following the OSI model. There are networks that were developed independently from the OSI model, most of which predate the model. TCP/IP, XNS and SNA, for example, are protocols that are not based on the OSI model.

At the present moment, many international standards have been developed for the OSI services and protocols (among which Formal Description Techniques) and there is much international activity in standardizing OSI conformance test suites.

1.3 Formal Description Techniques

Over the past years a number of languages (Formal Description Techniques or FDTs) have been developed to replace the natural language specification techniques. FDTs are used for different purposes, some are utilized for describing the behaviour of distributed systems (e.g., LOTOS, Estelle and SDL) and some are used as formal descriptions for test suites (e.g., TTCN). Using FDTs:

a) It is possible to describe systems precisely and unambiguously and to formally extract some properties of a system before its implementation.
b) The design process can be facilitated since most FDTs are based on precise, yet powerful and flexible mathematical models that offer a wide range of predefined symbols and operators.
c) In the case where the FDT is executable, the formal description can be used as a prototype of the entity specified and to allow to automate the generation of implementations.
d) It is possible to apply formal test generation methods.

1.4 LOTOS

LOTOS (Language Of Temporal Ordering Specifications) is a formal description technique standardized for the OSI services and protocols. LOTOS specifications describe distributed systems by defining the temporal relations among the interactions that represent the systems's externally observable behaviour [ISO 8807].

LOTOS specifications consist of two components: 1) a *control component* based on Milner's Calculus of communicating Systems (CCS) [Mil 80] and Hoare's Communicating Sequential Processes (CSP) [Hoa 85], which deals with the description of processes behaviours and interactions, and 2) a *data component* based on the formal theory of algebraic abstract data types ACT ONE [EM 85], that describes the data structures and value expressions. In the next section we will give an overview of both components of LOTOS. We will concentrate on the control component, and briefly outline the data component as it is outside the scope of this thesis. A detailed introduction to LOTOS can be found in [BB 87] and [LFH 92].

1.4.1 LOTOS data types

The requirement of abstraction from implementation details is one of the main objectives of FDTs. For this reason, LOTOS has 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:

a) reference to previously defined specifications in a library.
b) combinations and extensions of already existing specifications.
c) renaming and parameterization of specifications.
d) actualization of parameterized specifications.

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

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)), ...*

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.

1.4.2 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 [g1, g2, ... gn] (v1, v2, ... vm): 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 means of combining LOTOS operators, constants 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 values to be accepted. An experiment 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 a process that 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:

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

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 1.2 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.

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_1	g ?x_2:s_2	$s_1 = s_2$	value generation	after synchronization $x_1 = x_2 = x$ $x \in$ domain(s_1)

Table 1.2 : Types of interaction

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

In the following, we present the syntax and semantics of LOTOS behaviour expressions. We recall that a precise definition of the syntax and semantics of LOTOS is given in [ISO 8807].

Inaction

In LOTOS, a process can be in a situation 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.

Action prefix

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

$$B = g\ d_1 d_2 \ldots d_n\ [P];\ B'$$

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

$$B = i;\ B'$$

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:

$$B = B_1\ []\ B_2$$

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.

$$B = B_1\ |[g_1, \ldots\ g_m]|\ B_2$$

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

$$B = B_1 \; ||| \; B_2 = B_1 \; |[\;]| \; B_2 \text{ (Interleaving)}$$
$$B = B_1 \; || \; B_2 = B_1 \; |[g_1, \ldots g_m]| \; B_2 \text{ (Full synchronization)}$$

In the first case, the synchronization set is empty while in the second case it contains all possible gates of both behaviours.

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 hidden to the environment.

$$B = \textbf{hide } g_1, \ldots g_m \textbf{ in } B'$$

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.

Successful termination

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

$$B = \textbf{exit}(E_1, \ldots E_m)$$

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

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.

$$B = B_1 >> B_2$$

In general, enabling is associated with the passing of parameters that are necessary for the enabled behaviour by means of the *exit* and *accept* operators.

$$B = \ \ldots\ \textbf{exit}(x_1, \ldots x_n)\ >>\ \textbf{accept}\ \ y_1{:}s_1, \ldots y_n{:}s_n\ \ \textbf{in}\ \ B'$$

B is enabled with $x_1, \ldots x_n$ as values for $y_1, \ldots y_n$.

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.

$$B = [Guard] \rightarrow B'$$

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

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 $P[h_1, \ldots h_n]$ $(s_1, \ldots s_m)$ be a process.

$$B = P[g_1, \ldots g_n]\ (t_1, \ldots t_m)$$

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

1.5 Conformance testing

The goal of conformance testing is to check whether a protocol implementation conforms to the related protocol specification [ISO 9646]. This can be done by running a set of test cases (a test suite) on the implementation under test (IUT). The main problems in conformance testing are: the formalization of the conformance relation (between a specification and an IUT), test generation, test selection, distributed testing and analysis of results.
The introduction of FDTs made it possible to formalize these problems, since the specification is now written using a formal language that has precise semantics and syntax [BALT 90]. Basing testing on FDTs has many advantages:

a) The relation between a specification and a conforming implementation can be described by a sound formal definition.

b) Based on this definition, algorithms can be developed to derive tests from formal specifications.

c) It is possible to define a precise mathematical measure of the extent to which products have been tested (test coverage).

d) Tools can be developed to aid testing such as tools for test derivation and test selection from formal specifications.

1.5.1 Test derivation

Test derivation addresses the problem of getting tests or experiments from the specification. Currently, this procedure is done manually and so it is time consuming and error prone. But as mentioned before, test derivation can be automated by using FDTs. As shown in Figure 1.1, test derivation occupies a central role in conformance testing.

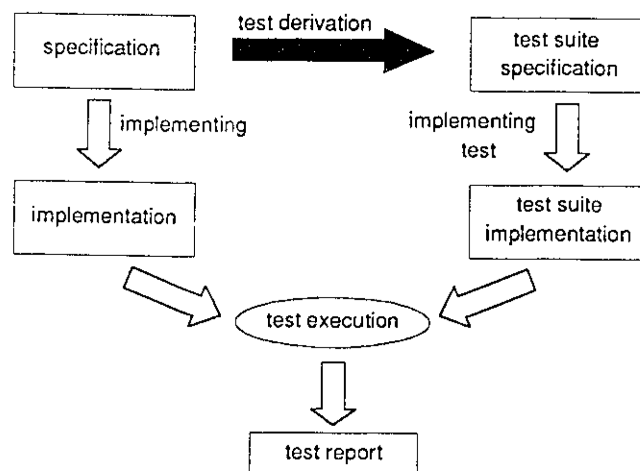


Figure 1.1 : The importance of test derivation

The test suite specification is usually written in a standardized language suitable for describing test suites. TTCN (Tree and Tabular Combined Notation) is best suited for that purpose [PM 90].

1.5.2 Test selection

The set of test cases necessary to completely test an implementation may be infinite. In practice however, in order to test an implementation in a reasonable amount of time and consuming only a reasonable amount of resources, the number of test cases in a test suite has to be within reasonable limits. Therefore, *selection criteria* have to be defined in order to reduce the set of test cases [Mye 79]. One way of evaluating test selection criteria is by means of determining the *value* and the *cost* of a test suite. The *value* of a test suite can be formulated as a function which relates the test suite and the specification to a value. This function should reflect the *quality* of a test. It should also include informal notions such as "coverage" or "error detection power". The *cost* can be formulated as a function on a test suite. An "infinite" test suite has an infinite *cost*. A "finite" test suite has a finite *cost* that could still be very high. The *costs* of a test suite are directly dependant on reasonable physical aspects, such as the number of test cases, the average length of a test case, the execution time, etc ... [BALT 90]. □

Note that there are many aspects of conformance testing which we do not address in the thesis:

- From abstract tests to executable tests.
- Test purposes and test coverage.
- Test selection.
- Inconclusive verdicts.
- Robustness testing.
- Conformance requirements, PICS, PIXIT ...

These are indeed important subjects in conformance testing. However, each one of them is quite complex in its own and covering them all would require considerable work.

The relation between the LOTOS testing theory and ISO conformance testing [ISO 9646] has been explored in [BALT 90] and [WBL 91].

1.6 Motivation of the thesis

Test derivation methods can be classified into two main groups: methods that derive tests directly from the formal description (FD) and methods that consist of transforming the FD to a model in another formalism and subsequently derive tests from the model. The first class is referred to as *direct derivation techniques*, the second as *indirect derivation techniques* (see Figure 1.2).

The transformation to the intermediate model may preserve all the information in the FD or may not; in the latter case it should preserve the necessary information for deriving the test suite. Examples of these intermediate models are: Finite State Machines (FSMs), Labelled Transition Systems (LTSs), Parameterized Labelled Transition Systems (pLTSs), Data Flow Charts, Petri Nets, Execution Trees ...

At the university of Ottawa, several tools have been developed for the LOTOS FDT, among which are ISLA [HH 89] and SELA [Ash 92]. The tool SELA in conjunction with ISLA can transform a LOTOS specification into a *symbolic execution tree* (also referred to as a *behaviour tree*). This model can be useful for test derivation purposes. Indeed, most of the test derivation theory and algorithms are developed for the LTS model which is similar if not identical to the symbolic execution tree model.

Most test derivation tools that have been developed for LOTOS belong to the direct test

derivation class. Due to the complexity of LOTOS, the implementation of these tools is hard and imposes certain restrictions on the formal specification. One of the major restrictions is the handling of *infinite behaviours* including recursion (see Chapter 3). The simplicity of the symbolic execution tree model and its ability to express infinite behaviours and to preserve all the information necessary for test derivation gave us the motivation to adapt the current test derivation theory for that model and to implement a new tool for test suite generation (indirect test derivation) based on it.

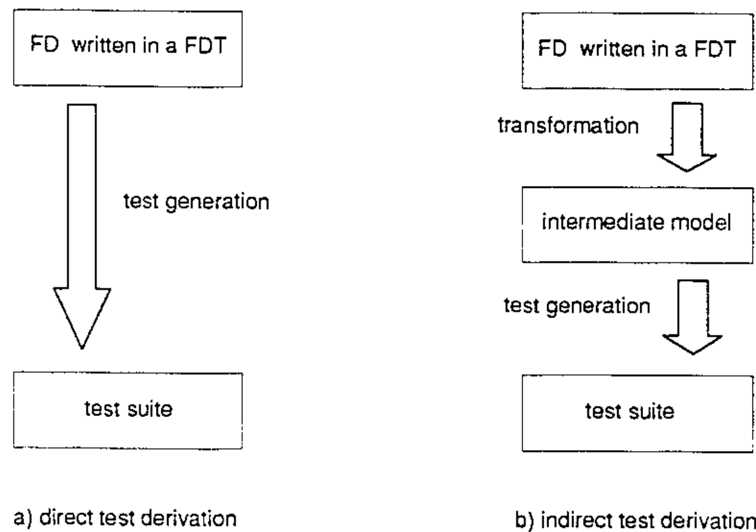


Figure 1.2 : Direct and indirect test derivation

1.7 Summary of results

In this thesis, we presented the CO-OP method [Wez 88] and we generalized it to treat infinite behaviours, by adapting the indirect test derivation methodology. We also described algorithms for proving conformance by using *complete failure trees*.

We have implemented all the new ideas found in the thesis in an interactive tool. Unlike most of the current tools, our tool LOTEST is capable of treating infinite behaviours and a large subset of Full LOTOS specifications.

With respect to the complexity of problems in real-life conformance testing, our contribution is modest indeed, however we hope to have made some contributions to the theory, and helped the work of researchers and students who will study the LOTOS model for conformance testing by using our tool.

1.8 Organization of the thesis

In chapter 2, we present the current test derivation theory for LTSs; in particular we will introduce the CO-OP method [Wez 88]: a test derivation algorithm for FDTs whose semantics are based on the LTS model.

In chapter 3, we present our variation on the CO-OP method: an adaptation of the method that makes the algorithm suitable to treat infinite behaviours. We will give more details on how we applied the method on the symbolic execution tree model.

In chapter 4, we introduce another application for symbolic execution trees in conformance testing: building *complete failure trees* to *prove conformance* in some special cases.

In chapter 5, we present an adaptation of the CO-OP method to treat special cases of symbolic execution trees built from Full LOTOS specifications.

In chapter 6, we give some details on implementing the various ideas and algorithms found in this thesis.

In appendix A, we present LOTEST (a tool for LOTOS testing theory): the implementation of the various ideas and algorithms found in this thesis.

In appendix B, we present samples of output from the tool LOTEST, to show to what extent our work can be practical in real life applications.

Chapter 2

A Formal Approach To Conformance Testing

The semantics of basic LOTOS, as well as many other languages, e.g., CCS [Mil 80, Mil 89], CSP [Hoa 85], ACP [BeKl 85], are defined in terms of *Labelled Transition Systems* (LTSs), so solving the problem of test derivation for LTSs also solves the problem for LOTOS behaviours.

2.1 Labelled transition systems

Definition 2.1 **(labelled transition systems)**

A labelled transition system (LTS) is a 4-tuple $<$ Stat, Act, Trans, $s_0 >$, where

1. Stat is a countable non empty set of states.
2. Act is a set of actions.
3. Trans = $\{ <s_1, a, s_2> \mid s_1, s_2 \in$ Stat and $a \in$ Act $\}$ is a set of transitions.
4. s_0 is the initial state. □

A transition in a LTS is a 3-tuple $< s_1, a, s_2 > \in$ Trans, usually represented by: $s_1 -a\rightarrow s_2$,

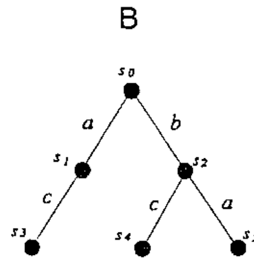
and we have by definition: $< s_1, s_2 > \in -a\rightarrow$.

$-a\rightarrow =_{def} \{ <s_1, s_2> | s_1, s_2 \in$ Stat and $<s_1, a, s_2> \in$ Trans $\}$.

The set Act includes the unobservable (or internal) action i (also denoted by τ). The set Act-{i} (set of observable actions) is denoted by L.

Example 2.1

Consider the following LTS:



We have:

Stat = $\{s_0, s_1, s_2, s_3, s_4, s_5\}$

Act = {a, b, c, i}

L = {a, b, c}

Trans = $\{ <s_0, a, s_1>, <s_0, b, s_2>, <s_1, c, s_3>, <s_2, c, s_4>, <s_2, a, s_5> \}$

$-a\rightarrow = \{ <s_0, s_1>, <s_2, s_5> \}$

$-b\rightarrow = \{ <s_0, s_2> \}$

$-c\rightarrow = \{ <s_1, s_3>, <s_2, s_4> \}$ □

Table 2.1 contains the list of symbols and notations for labelled transition systems we will be using in this thesis.

Notation	Meaning
L^*	is the set of all strings of observable actions
ε	is the empty string
$\eta, \eta_1, \ldots \eta_n$	are actions from the set Act
$\mu, \mu_1, \ldots \mu_n$	are actions from the set L
$B, C, B_1, \ldots B_n$	are labelled transition systems
σ, σ'	are strings of actions from the set L
$B{-}\eta{\rightarrow}C$	$B = \langle Stat, Act, Trans, s_B \rangle$; $C = \langle Stat, Act, Trans, s_C \rangle$ and $\langle s_B, \eta, s_C \rangle \in Trans$
$B{-}\eta{\rightarrow}$	$\exists C : B{-}\eta{\rightarrow}C$
$B{-}\!\!\!/\,\eta{\rightarrow}$	$\nexists C : B{-}\eta{\rightarrow}C$
$B{-}\eta_1 \ldots \eta_n{\rightarrow}C$	$\exists B_i \ (1 \leq i \leq n) : B = B_0{-}\eta_1{\rightarrow}B_1{-}\eta_2{\rightarrow}\ldots{-}\eta_n{\rightarrow}B_n = C$
$B{-}i^0{\rightarrow}C$	$B = C$
$B{=}\varepsilon{\Rightarrow}C$	$\exists n \geq 0 : B{-}i^n{\rightarrow}C$
$B{=}\mu{\Rightarrow}C$	$\exists B_1, B_2 : B{=}\varepsilon{\Rightarrow}B_1{-}\mu{\rightarrow}B_2{=}\varepsilon{\Rightarrow}C$
$B{=}\mu_1 \ldots \mu_n{\Rightarrow}C$	$\exists B_i \ (1 \leq i \leq n) : B = B_0{=}\mu_1{\Rightarrow}B_1{=}\mu_2{\Rightarrow}\ldots{=}\mu_n{\Rightarrow}B_n = C$
$B{=}\mu_1 \ldots \mu_n{\Rightarrow}$	$\exists C : B{=}\mu_1 \ldots \mu_n{\Rightarrow}C$
$B{\neq}\sigma{\Rightarrow}$	$\nexists C : B{=}\sigma{\Rightarrow}C$
$B{=}\sigma{\rightarrow}C$	if $\sigma = \varepsilon$ then $B = C$ if $\sigma = \sigma'\mu$ then $\exists B' : B{=}\sigma'{\Rightarrow}B'{-}\mu{\rightarrow}C$
$\Sigma_{j \in \{1, \ldots n\}} B_j$	$B_1 \, [] \, B_2 \, [] \, \ldots \, B_n$
Out(B)	$\{ a \in L \mid B{=}a{\Rightarrow} \}$
Tr(B)	$\{ \sigma \in L^* \mid B{=}\sigma{\Rightarrow} \}$: the set of traces of B
stable(B)	$B{-}\!\!\!/\,i{\rightarrow}$
B refuses (σ, A)	$\exists B' : (B{=}\sigma{\Rightarrow}B'$ and $\forall a \in A : B'{\neq}a{\Rightarrow})$

Table 2.1 : Notations for labelled transition systems

2.2 The implementation relation

We will use the notion of conformance as defined in [Bri 88] where conformance is defined by an implementation relation *conf*.

Definition 2.2 **(conformance relation)**

$$I \text{ conf } S \Leftrightarrow \forall \sigma \in Tr(S), \forall A \subseteq L : \quad \text{if } I \text{ refuses}(\sigma, A) \text{ then } S \text{ refuses}(\sigma, A).$$

Where in general, the expression B *refuses*(σ, A) says that B refuses the set of actions A after the trace σ (see Table 2.1). □

It is stated in [Bri 87] that *I conf* S is a testable property only in the case of *non-diverging* specifications (i.e. do not contain infinite sequences of internal actions). It is shown that this property can be tested by a test suite $\Pi_{conf}(S)$ consisting of one element called the *canonical tester*. A formal definition of a canonical tester will be given later in the chapter.

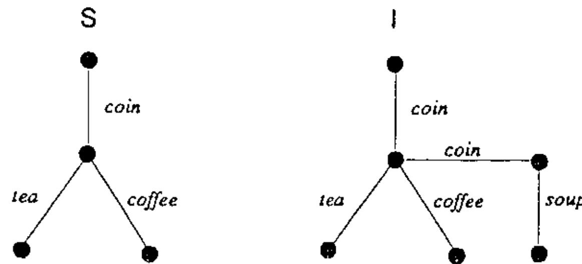
The *conf* relation is not transitive; so if S_3 *conf* S_2 and S_2 *conf* S_1, we do not have necessarily S_3 *conf* S_1.

We can also see that the *conf* relation doesn't test for robustness: it allows the implementation I to contain traces outside the set Tr(S). The reason for this is that robustness tests must include traces that are not part of the specification. It is not clear how these tests can be derived from specifications; indeed, the number of these tests can be very large even in the case of finite processes.

The following example illustrates that situation:

Example 2.2

Consider the following two LTSs:



Although I contains traces outside Tr(S), according to definition 2.2, I conf S. □

One can define a stronger relation *red* that doesn't allow implementations to contain traces outside the traces of the specification [Led 90]:

Definition 2.3 (reduction relation)

I red S $\Leftrightarrow$ $\forall \sigma \in L^*, \forall A \subseteq L$: if I refuses(σ, A)
then S refuses(σ, A). □

As opposed to the *conf* relation, the *red* relation is transitive and a preorder [Led 90]. The *testing equivalence* relation denoted by $\approx$ can be defined in terms of the *red* relation.

Definition 2.4 (testing equivalence)

$B_1 \approx B_2 \Leftrightarrow B_1$ red B_2 and B_2 red B_1. □

As we've seen in the previous definitions, the informal concept of "implementation" can be defined in several ways. An implementation can be seen as a reduction of a given specification or it can be interpreted by means of the conformance relation. An implementation can also be seen as a "real/physical system", that is a relation between an abstract description and a real system [BSS 87].

2.3 Canonical testers

The view we take in this thesis is that a conformance tester is a process running concurrently with the process under test. This is an abstraction of the implemented tester. For the conformance tester the concept of *general canonical testers* introduced in [Bri 87] is used. Informally, a general canonical tester T(S) of a specification S is a process that is (i) capable of exploring all and only traces in S, and (ii) must satisfy the following property: for any process P, P conforms to S iff every deadlock between P and the tester T(S) can be explained by the tester reaching a terminal state.

Since canonical testers are capable of exploring all traces in S, they are viewed as the specification of an exhaustive test suite. In practice the canonical tester is not used directly for testing; but *test cases* are derived from it. Test cases are taken to be *reductions of the canonical tester*. Irreducible test cases are called *basic test cases*. Later in the chapter rules for extracting test cases from the canonical tester will be given. Building the canonical tester is nothing more than an intermediate step before deriving test cases. As we will see later in the chapter, deriving test cases from the canonical tester is easier than deriving them directly from the specification.

Running a test case on an Implementation Under Test (IUT), and having the test case pass successfully, never proves in most cases that the IUT conforms to its specification but having the test case fail, proves that the IUT is not a conformable implementation.

Remark

The Testing Architecture we use is the simplest (see Figure 2.1). Using LOTOS terminology, we suppose that the tester plays the role of the *environment* for the IUT and synchronizes with all observable actions of the IUT. The actions of the tester are close and *direct* to the IUT as opposed to *remote* via a communication channel.

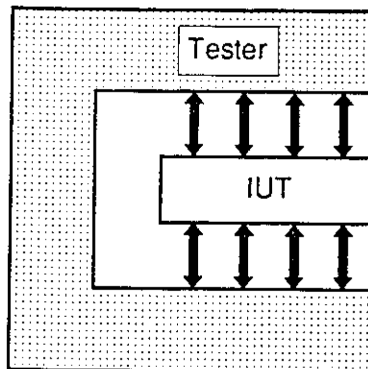


Figure 2.1 : The Utilized Testing Architecture

Using ISO-OSI terminology and knowing that OSI protocol standards define the allowed behaviour of a *protocol entity* (i.e. the dynamic conformance requirements) in terms of the *protocol data units* (PDUs) and both the *abstract service primitives* (ASPs) above and below that entity [ISO 7498], we restrict our interest to the *local test methodology* which uses control and observation of the ASPs directly above and below the entity under test. The other OSI test methodologies are: the *distributed test methodology* and the *remote test methodology*. The testing architecture we use is not the most suitable for real life conformance testing, since it assumes synchronous communication between the tester and the IUT. An asynchronous model was proposed in [TrVe 92]. □

Formally a general canonical tester is defined as follows:

Definition 2.5 (general canonical tester)

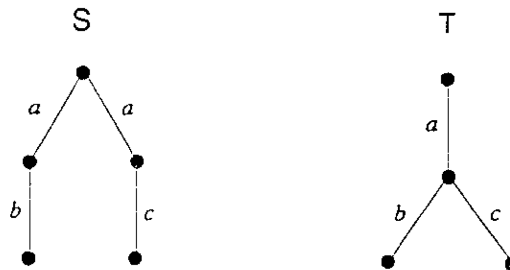
Let S be a process. A (general) canonical tester of the process specification S is a process T(S) that satisfies:

(i) Tr(T(S)) = Tr(S); and

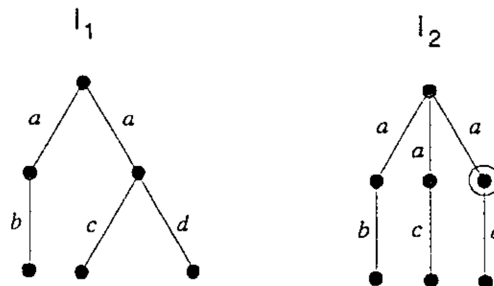
(ii) For all processes P, P conf S iff
for all $\sigma \in L^*$, P||T(S) $=\sigma\Rightarrow$ B and B $\approx$ stop implies T(S) $=\sigma\Rightarrow$ B′ and B′ $\approx$ stop

Example 2.3

Consider the following two LTSs:



T is a canonical tester of S. Later in this chapter we will present the CO-OP method: an algorithm that generates canonical testers for any LTS. T was created using that method. Now, consider these two LTSs:



One can see that I_1 conforms to S, since by fully synchronizing I_1 and T, a deadlock occurs only when T reaches a terminal state. I_2 doesn't conform to S, since a deadlock may occur between I_2 and T at a point where T hasn't reached yet a terminal state e.g., when I_2 moves to the state marked by a circle.

2.4 The CO-OP method

The CO-OP method, introduced in [Wez 88] and [Wez 89], is a method for deriving canonical testers from LTSs; the method is based on ideas found in [Ste 86], [Pit 87] and [FP 88]. The CO-OP method makes it possible to derive testers compositionally from basic LOTOS behaviours, that is the tester of $B_1 * B_2$ can be constructed from the testers of B_1 and B_2 where * is any Basic LOTOS operator. It was also proven that, in general, $T(B_1 * B_2) \neq T(B_1) * T(B_2)$.

Three attributes play a major role in the method: the *compulsory set*, the *options set* and the expression *B after* μ.

2.4.1 Compulsory and options sets

States of a LTS can be categorized into two groups:

- Stable states that accept only external actions and cannot do an internal action. The tester should *always* offer the possibility of interacting in at least one action that can be performed in a stable state. If not the tester may deadlock with a correct implementation. The compulsory set will contain those actions.
- Unstable states that may accept internal actions. These actions, having the possibility to be rejected by the specification itself, *may* be used in a test case so they are included in the options set.

Formally, both sets can be defined as follows:

Definition 2.6 (compulsory and options sets)

Let B be a LTS.

Compulsory(B) = $\{ A \subseteq L \mid \exists B' : B{=}\varepsilon{\Rightarrow}B'{-}i{\not\rightarrow} \text{ and } A = \{ a \in L \mid B'{-}a{\rightarrow} \} \}$

Options(B) = $\{ a \in L \mid \exists B' : B{=}\varepsilon{\Rightarrow}B'{-}i{\rightarrow} \text{ and } B'{-}a{\rightarrow} \}$ □

The following proposition provides more convenient definitions for the same concepts.

Proposition 2.1

Let B be a LTS.

a) Compulsory(B) = $\{ A \subseteq L \mid \exists B' : B{=}\varepsilon{\Rightarrow}B' \text{ and stable}(B') \text{ and } A = \text{Out}(B') \}$

b) Options(B) = $\bigcup_{B' : B=\varepsilon\Rightarrow B' \text{ and } \neg \text{stable}(B')} \{ a \in L \mid B'{-}a{\rightarrow} \}$

Proof

The proof of a) and b) follows from the definitions of stable(B) and Out(B) (see Table 2.1). By replacing each of the terms stable(B′) and Out(B′) in a) and b) with the corresponding definition we obtain definition 2.6. □

For each set in the compulsory set, at least one element must be used in a test case so to avoid a deadlock with a conforming implementation. For that, the method needs to compute the set of all sets that can be formed by choosing precisely one member of each element of the compulsory set. The set obtained is called the orthogonal of the compulsory set. Formally, the orthogonal set is defined as follows:

Definition 2.8 (orthogonal set)

Given a set $M \subseteq \{ A \subseteq L \mid A \neq \emptyset \}$, Orth(M) is defined to be the set of all sets which can be formed by choosing exactly one member of each element of M [Pit 87].

2.4.2 The expression *B after μ*

Building the canonical tester T of a LTS B using the CO-OP method is a recursive procedure involving building the initial behaviour of T from the initial behaviour of B, then, calling the same procedure for the expressions *B after μ* where $\mu \in Out(B)$.

Definition 2.7 (*B after μ*)

Let B be a LTS.

$B \text{ after } \mu \;=_{def}\; \Sigma_{B' : B = \mu \rightarrow B'}\; i; B'$

2.4.3 Initial behaviour of the tester

Given the initial behaviour of a process B, the CO-OP method computes the initial behaviour of the tester as follows:

if $\emptyset \notin$ Compulsory(B) then

$$T(B) \;=\; \sum_{V \in Orth(Compulsory(B))} i; \Sigma_{a \in V}\, a; \ldots$$
$$[\,]\quad \sum_{a \in Options(B)} a; \ldots$$

else $\{ \emptyset \in$ Compulsory(B) $\}$

$$T(B) \;=\; i; stop$$
$$[\,]\quad \sum_{a \in Out(B)} a; \ldots$$

As we see in the formula, a special case has to be distinguished: the case when the

compulsory set contains the empty set. By looking at the definition of the compulsory set, we can see that it happens if and only if B has the possibility to deadlock with any environment (B ≈ stop). Although in many cases the occurrence of deadlock is a design fault, for that case the tester should also have the ability to deadlock to avoid rejecting conforming implementations.

2.4.4 Building the canonical tester

The tester has the following recursively defined behaviour expression:

if $\emptyset \notin$ Compulsory(B) then

T(B) = $\sum_{V \in Orth(Compulsory(B))}$ i; $\sum_{a \in V}$ a; T(B after a)

[] $\sum_{a \in Options(B)}$ a; T(B after a)

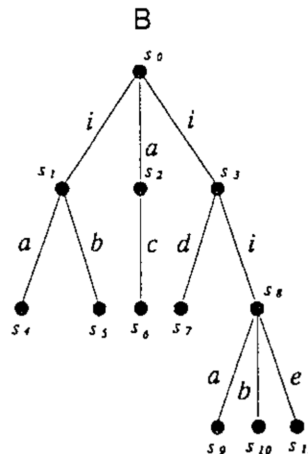
else { $\emptyset \in$ Compulsory(B) }

T(B) = i; stop

[] $\sum_{a \in Out(B)}$ a; T(B after a)

Example 2.4

Consider the following LTS:



The set of stable states of B is: $\{s_1, s_2, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}\}$

The set of unstable states of B is: $\{s_0, s_3\}$

B after a = i; stop [] i; c; stop [] i; stop ≈ i; stop [] i; c; stop = B_1

B after b = i; stop [] i; stop ≈ i; stop = B_2

B after d = i; stop = B_2

B after e = i; stop = B_2

Compulsory(B) = {{a, b}, {a, b, e}}

Orth(Compulsory(B)) = {{a}, {a, b}, {a, e}, {b, a, e}}

Options(B) = {a, d}

Out(B) = {a, b, d, e}

T(B) = i; a; T(B_1)
[] i; (a; T(B_1) [] b; T(B_2))
[] i; (a; T(B_1) [] e; T(B_2))
[] i; (a; T(B_1) [] b; T(B_2) [] e; T(B_2))
[] a; T(B_1) [] d; T(B_2)

Next we proceed to calculate T(B_1)

B_1 after c = i; stop = B_2

Compulsory(B_1) = {Ø, {c}}

Options(B_1) = Ø

Out(B_1) = {c}

T(B_1) = i; stop Since Ø ∈ Compulsory(B_1)
[] c; T(B_2)

Finally the calculation of T(B_2)

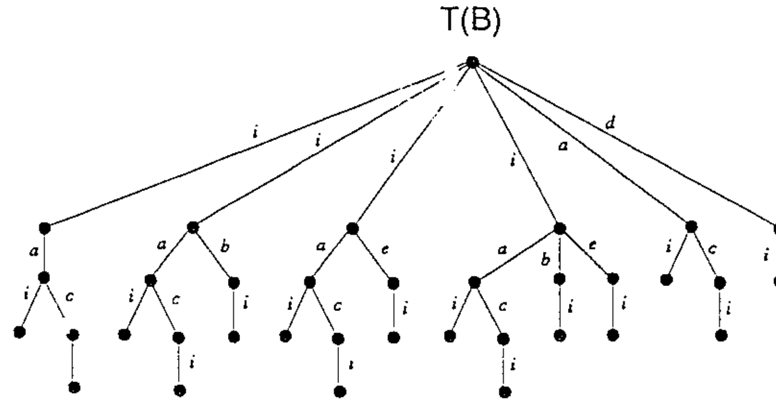
Compulsory(B_2) = {Ø}

Options(B_2) = Ø

Out(B_2) = Ø

$T(B_2)$ = i; stop Since $\emptyset \in$ Compulsory(B_2)

Finally we get:



By looking at the Tester we notice a lot of redundancies. In the next section we present a number of rules for eliminating some of these redundancies.

2.4.5 Optimizing the canonical tester

Three rules are applied to remove some of the redundancies [Wez 88]:

Rule 1: if $\exists\ C_1, C_2 \in$ Compulsory(B) such that $C_1 \subseteq C_2$ and $C_1 \neq \emptyset$ then
Options(B) $\leftarrow$ Options(B) $\cup$ $\{C_2 - C_1\}$
Compulsory(B) $\leftarrow$ Compulsory(B) - $\{C_2\}$

Rule 2: if $\exists$ a $\in$ Options(B), C $\in$ Compulsory(B) such that a $\in$ C then
Options(B) $\leftarrow$ Options(B) - {a}

Rule 3: if $\exists\ O_1, O_2, O_3 \in$ Orth(Compulsory(B)) such that $O_3 = O_1 \cup O_2$ then
O_3 can be removed. The resulting set is denoted by Orth_c(Compulsory(B)).

In chapter 3 more rules will be given for eliminating some of the redundant internal actions in the canonical tester.

Example 2.5

Applying Rules 1, 2 and 3 to the LTS in example 2.4 we get:

Compulsory(B) = {{a, b}} Rule 1

Orth(Compulsory(B)) = {{a}, {b}}

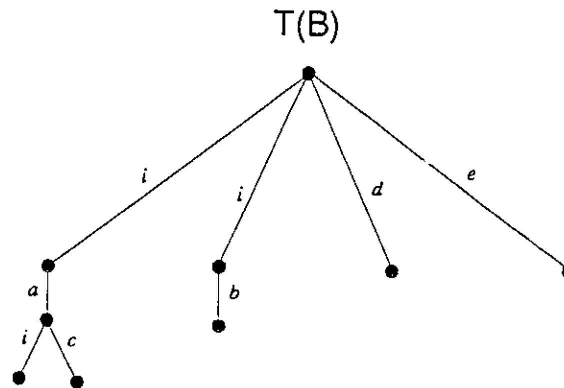
Options(B) = {d, e} Rules 1 and 3

T(B) = i; a; $T(B_1)$

[] i; b; $T(B_2)$

[] d; $T(B_2)$ [] e; $T(B_2)$

Removing the redundant internal actions we get:



2.4.6 Extracting test cases from the canonical tester

The canonical tester can be viewed as the specification of a test suite; a test suite that tests for the *conf* relation must conform to the canonical tester. Rules for extracting test cases from the canonical tester have to be defined. Building the canonical tester is only an intermediate step before building the set of test cases making up the test suite.

A test case is viewed as a *reduction of the canonical tester* [Bri 88] that preserves property (ii) of canonical testers; i.e. every deadlock between the test case and the process under test, without having the test case reached a terminal state, implies that the process under test does not conform to the specification. Test cases do not have to satisfy property (i) of canonical testers stating that the set of traces of the canonical tester must be equal to the set of traces of the specification; instead, the set of traces of a test case must be a subset of the set of traces of the specification.

Extracting test cases from canonical testers built using the CO-OP method is a very easy task. As we have seen, canonical testers have the following form:

$$\begin{array}{lll} T & = & \sum_i i;\ \sum_j a_{ij};\ T_{ij} \\ & [] & \sum_k b_k;\ T_k \end{array}$$

where the T_{ij}'s and the T_k's have the same form as T.

Let's recall that a test case is said to be *basic* if it can't be reduced to form other test cases; i.e. a basic test case is an *irreducible reduction* of the canonical tester [Bri 89]. Extracting a basic test case from T can be done by selecting an i-branch from T, removing the *i*, selecting *all* the corresponding a_{ij}'s. Any number of actions b_k can be optionally selected, though causing the test case to be no longer basic. If there are no i-branches all actions b_k have to be selected. The same procedure is then applied to the T_{ij}'s and the T_k's.

This procedure can be made clearer by the following examples.

Example 2.6

Going back to example 2.5, basic test cases from T(B) are:

T_1 = a; stop

T_2 = b; stop

The following test cases are not basic:

T_3 = a; (i; stop [] c; stop)

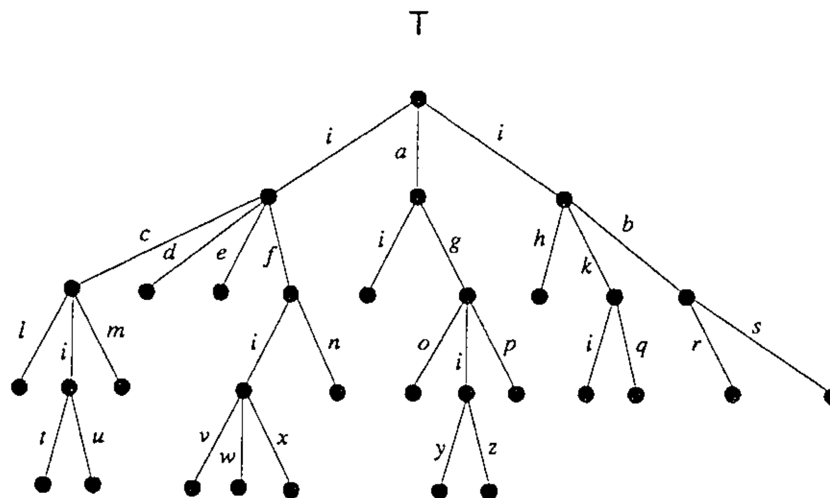
T_4 = a; (i; stop [] c; stop) [] d; stop [] e; stop

T_5 = b; stop [] d; stop [] e; stop □

The following is a more complex example:

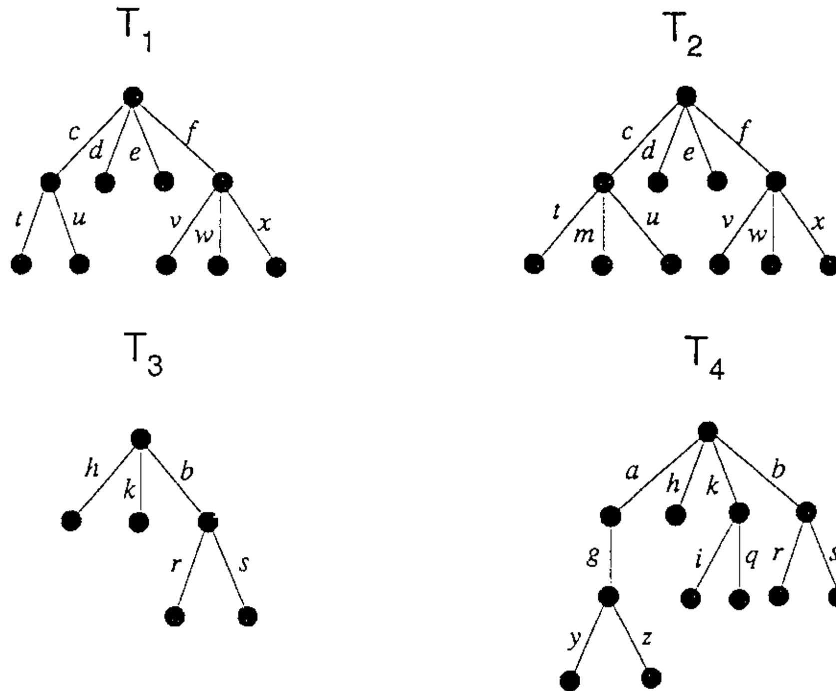
Example 2.7

Consider the following canonical tester:



We can see that T has the general format of a canonical tester derived using the CO-OP method.

Some test cases from T are:



T_1 and T_3 are basic test cases. T_2 and T_4 were obtained by adding optional actions to T_1 and T_3. In this example the set of optional actions is *{a, g, l, m, n, q}*. Optional actions are the actions selected from the *Options* sets or from the *Out* sets in the case when the *Compulsory* set contains the empty set (see section 2.4.3).

2.5 Chapter summary

In this chapter, we have presented an overview of the existing work in the theory of LOTOS conformance testing, in particular test derivation. We introduced the CO-OP method [Wez 88], an algorithm that generates canonical testers from finite Basic LOTOS behaviours. In the next chapter, we present our variation on the CO-OP method for treating infinite LOTOS behaviours.

Chapter 3

Generalization Of The CO-OP Method

After presenting an overview of the existing work done in the domain of conformance testing for basic LOTOS, we will introduce a variation of the CO-OP method that will eliminate some of the limitations of the current method. The generalized method will include an algorithm for generating canonical testers for recursive basic LOTOS behaviours. In this chapter we use additional notations from Table 3.1.

3.1 Critique of the current CO-OP method

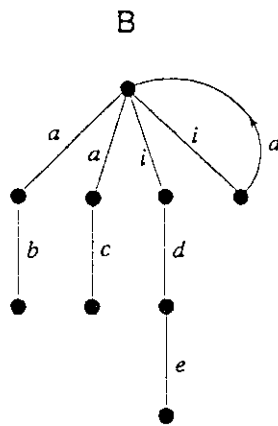
3.1.1 Recursive behaviour expressions

As we have seen in chapter 2, generating the canonical tester of a process B is a recursive process involving generating canonical testers of expressions of the form: B *after* μ where B *after* $\mu = \sum_{B' : B = \mu \rightarrow B'} i; B'$. This makes it hard to detect which expressions were already processed (the corresponding tester is generated), due to the fact that comparing these expressions is by itself a hard problem. In order to generate *complete canonical testers*, the method needs a simpler way of detecting recursion (cycles in the LTS).

Example 3.1

Consider the following recursive behaviour expression:

B = a; b; stop [] a; c; stop [] i; d; e; stop [] i; d; B



Compulsory(B) = {{d}}

Orth(Compulsory(B)) = {{d}}

Options(B) = {a}

T(B) = a; T(B after a) [] i; d; T(B after d)

B after a = B_1 = i; b; stop [] i; c; stop

B after d = B_2 = i; e; stop [] i; B

Compulsory(B_1) = {{b}, {c}}

Orth(Compulsory(B_1)) = {{b, c}}

Options(B_1) = ∅

$\text{Compulsory}(B_2) = \{\{d\}, \{e\}\}$

$\text{Orth}(\text{Compulsory}((B_2)) = \{d, e\}$

$\text{Options}(B_2) = \{a\}$

$T(B_1)$ = b; $T(B_1$ after b) [] c; $T(B_1$ after c)

$T(B_2)$ = i; (d; $T(B_2$ after d) [] e; $T(B_2$ after e)) [] a; $T(B_1)$

B_1 after b = stop

B_1 after c = stop

B_2 after d = B_3 = i; e; stop [] i; B

B_2 after e = stop

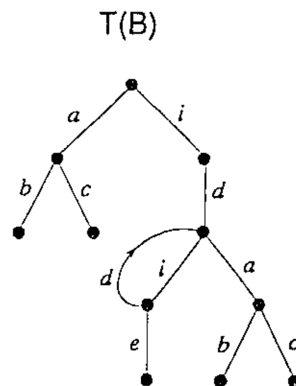
Since we can easily see that $B_3 = B_2$, we can stop the recursive process and write:

T(B) = a; $T(B_1)$ [] i; d; $T(B_2)$

$T(B_1)$ = b; stop [] c; stop

$T(B_2)$ = i; (d; $T(B_2)$ [] e; stop) [] a; $T(B_1)$

T(B)



In general, checking if two behaviour expressions are equal, is not as simple as in this example, especially if both behaviour expressions are non-deterministic.

Notation	Meaning
$s, s', s'', s_1, \ldots s_n$	are states from the set Stat
$s-\eta\rightarrow s'$	$< s, \eta, s' >$ is a transition
$s-\eta\rightarrow$	$\exists s' : s-\eta\rightarrow s'$
$s\not-\eta\rightarrow$	$\nexists s' : s-\eta\rightarrow s'$
$s-\eta_1\ldots\eta_n\rightarrow s'$	$\exists s_i\ (1 \leq i \leq n) : s = s_0-\eta_1\rightarrow s_1-\eta_2\rightarrow\ldots-\eta_n\rightarrow s_n = s'$
$s-i^0\rightarrow s'$	$s = s'$
$s=\varepsilon\Rightarrow s'$	$\exists n \geq 0 : s-i^n\rightarrow s'$
$s=\mu\Rightarrow s'$	$\exists s_1, s_2 : s=\varepsilon\Rightarrow s_1-\mu\rightarrow s_2=\varepsilon\Rightarrow s'$
$s=\mu_1\ldots\mu_n\Rightarrow s'$	$\exists s_i\ (1 \leq i \leq n) : s = s_0=\mu_1\Rightarrow s_1=\mu_2\Rightarrow\ldots=\mu_n\Rightarrow s_n = s'$
$s=\mu_1\ldots\mu_n\Rightarrow$	$\exists s' : s=\mu_1\ldots\mu_n\Rightarrow s'$
$s\neq\sigma\Rightarrow$	$\nexists s' : s=\sigma\Rightarrow s'$
$s=\sigma\rightarrow s'$	if $\sigma = \varepsilon$ then $s = s'$ if $\sigma = \sigma'\mu$ then $\exists s'' : s=\sigma'\Rightarrow s''-\mu\rightarrow s'$
Out(s)	$\{ a \in L \mid s=a\Rightarrow \}$
Tr(s)	$\{ \sigma \in L^* \mid s=\sigma\Rightarrow \}$
stable(s)	$s\not-i\rightarrow$
deadlock($\{s_1, s_2, \ldots s_n\}$)	$\nexists\mu : s_1=\mu\Rightarrow$ and $s_2=\mu\Rightarrow$ and ... $s_n=\mu\Rightarrow$
s refuses (σ, A)	$\exists s' : (s=\sigma\Rightarrow s'$ and $\forall a \in A : s'\neq a\Rightarrow)$

Table 3.1 : Other notations for labelled transition systems

3.1.2 Form of the canonical tester

By looking at the definition of the canonical tester (definition 2.5), we notice that it is not a LTS, since the 4-tuple < Stat, Act, Trans, s_0 > is not well defined. Indeed, the canonical tester is represented by a LOTOS-like expression that might be *infinite*. This second point is not as important as the first one, since a solution for the first problem, as we'll see later in this chapter, will solve the second problem.

The solution we propose will use the following idea:
Since the only way to detect recursion is to keep a database of the expressions *B after* μ, we propose a more efficient notation for representing the behaviour of a process after performing a certain action. To do this, we will use the concept of state sets.

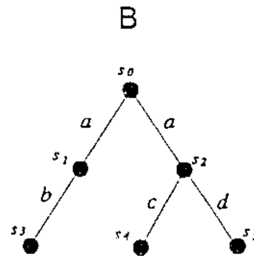
3.2 State set of a labelled transition system

Informally, the *state set* of a system is the state of the system as observed externally. Since a LTS might be non-deterministic; i.e. there could be s_1, s_2, s_3 and σ such that $s_1 {=} \sigma {\Rightarrow} s_2$, $s_1 {=} \sigma {\Rightarrow} s_3$ and $s_2 \neq s_3$, the state of the LTS might become unknown by the environment after a given trace of actions. Indeed, the only thing that the environment will be able to know is that the current state must be included in a certain set of states. This gives the clue of representing the state set of a system by a set that contains all the possible states of the system after a certain trace of actions. Going back to the idea formulated in the previous section, we can see that keeping a database of state sets is much simpler than keeping a database of the expressions *B after* μ, and it gives the same information (actually we can build from a state set the expression describing the behaviour of the system at that state).

We can illustrate this idea by the following example:

Example 3.2

Consider the following LTS:



After synchronizing with the environment on action a, the state of B becomes unknown to the environment; indeed the state of B can be either s_1 or s_2. By saying that the state set of B is the set $\{s_1, s_2\}$, we mean that B can make an internal decision to move to state s_1 or to state s_2. So formally a state set of a LTS can be defined as follows:

Definition 3.1 (state set)

Let B = < Stat, Act, Trans, s_0 > be a LTS.
A state set of B is any subset of S including the empty set. □

The expression *B after μ* plays an important role in the CO-OP method; a similar expression:
G after μ, where G is a state set of B, can be defined as follows:

Definition 3.2 (*G after μ*)

Let B = < Stat, Act, Trans, s_0 > be a LTS and G a state set of B. We have:
G after $\mu = \{ s \in Stat \mid \exists s' \in G : s' {=}\mu{\rightarrow} s \}$. □

Definition 3.2 can be extended to define expressions of the form *G after* $\mu_1\mu_2...\mu_n$.

Definition 3.3 (*G after* σ)

Let B = < Stat, Act, Trans, s_0 > be a LTS, σ a string of observable actions and G a state set of B.

G after σ can be defined recursively as follows:

Case 1) $\underline{\sigma = \varepsilon}$

G after σ = G

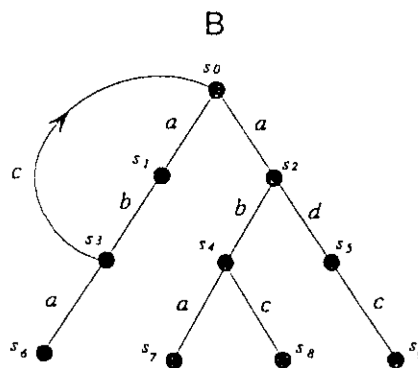
Case 2) $\underline{\sigma = \mu\sigma'}$

G after σ = (G after μ) after σ'

where *G after* μ is defined in 3.2.

Example 3.3

Consider the following LTS:



Suppose the initial state set is $G_0 = \{s_0\}$.

G_0 after a = $\{s_1, s_2\}$

G_0 after a b c = $\{s_0, s_8\}$

$\{s_0, s_4\}$ after a = $\{s_1, s_2, s_7\}$

G_0 after b = $\emptyset$ □

An equivalent definition of canonical testers (definition 2.5), using notations from Table 3.1, is given:

Definition 3.4 (general canonical tester)

Let B = < Stat, Act, Trans, s_0 > be a LTS, T = < Stat′, Act′, Trans′, t_0 > is said to be a general canonical tester of B iff

(i) Tr(B) = Tr(T); and

(ii) $\forall$P = < Stat″, Act″, Trans″, p_0 >, where P is a LTS, P conf B iff
$\forall \sigma \in L^*$, if $p_0 {=}\sigma{\Rightarrow} p_1$ and $t_0 {=}\sigma{\Rightarrow} t_1$ and deadlock($\{p_1, t_1\}$) then deadlock($\{t_1\}$)

3.3 The generalized CO-OP method for Basic LOTOS

3.3.1 Goal of the new method

The goal of the new method is to derive canonical testers from a finite state LTS. The method has to fully support recursion (cycles in the LTS); i.e., if the number of states of the LTS is finite, the number of states of its tester must also be finite. The derived canonical tester must be a LTS, i.e., the set of states and transitions must be defined clearly.

Formally the problem is:
Given a LTS B = < Stat, Act, Trans, s_0 > derive its general canonical tester T(B) a LTS that satisfies properties (i) and (ii) of definition 2.5.

The only restrictions imposed on B are:

a) The sets Stat, Act and Trans must be finite.

b) There is no cycle of internal actions in B: $\nexists s \in$ Stat : $s{-}i^n{\rightarrow}s$ and $n > 0$.

Restriction b) is also present in the test derivation theory for labelled transition systems.

3.3.2 Compulsory and options sets for the generalized CO-OP method

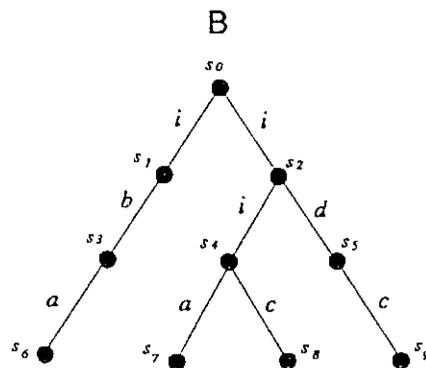
As in the original CO-OP method the compulsory and the options sets play a major role. The definition of both sets is almost identical to the one given previously, except that both sets are now applied to the state sets of a LTS instead of being applied to the LTS itself.

Definition 3.5 (compulsory and options sets)

Let B = < Stat, Act, Trans, s_0 > be a LTS and G a state set of B.
Compulsory(G) = { A ⊆ L | ∃g ∈ G and s ∈ Stat : g=ε⇒s and stable(s) and A = Out(s) }
Options(G) = { a ∈ L | ∃g ∈ G and s ∈ Stat : g=ε⇒s and ¬ stable(s) and s−a→ }

Example 3.4

Consider the following LTS:



Compulsory({s_0}) = {{b}, {a, c}}
Options({s_0}) = {d}
Compulsory({s_3}) = {{a}}

$\text{Options}(\{s_3\}) = \emptyset$

$\text{Compulsory}(\{s_0, s_3\}) = \{\{a\}, \{b\}, \{a, c\}\}$

$\text{Options}(\{s_0, s_3\}) = \{d\}$

3.3.3 Finding the compulsory and the options sets

Going back to example 3.4, we notice that:

$\text{Compulsory}(\{s_0, s_3\}) = \text{Compulsory}(\{s_0\}) \cup \text{Compulsory}(\{s_3\})$ and

$\text{Options}(\{s_0, s_3\}) = \text{Options}(\{s_0\}) \cup \text{Options}(\{s_3\})$

This result can be generalized by the following proposition:

Proposition 3.1

Let $B = < \text{Stat}, \text{Act}, \text{Trans}, s_0 >$ be a LTS and G a state set of B.

$\text{Compulsory}(G) = \bigcup_{g \in G} \text{Compulsory}(\{g\})$

$\text{Options}(G) = \bigcup_{g \in G} \text{Options}(\{g\})$

Proof

From definition 3.4 we have:

$$\begin{aligned}
\text{Compulsory}(G) &= \{ A \subseteq L \mid \exists g \in G \text{ and } s \in \text{Stat} : g{=}\varepsilon{\Rightarrow}s \text{ and stable}(s) \text{ and } A = \text{Out}(s) \} \\
&= \bigcup_{g \in G} \{ A \subseteq L \mid \exists s \in \text{Stat} : g{=}\varepsilon{\Rightarrow}s \text{ and stable}(s) \text{ and } A = \text{Out}(s) \} \\
&= \bigcup_{g \in G} \text{Compulsory}(\{g\}) \\
\text{Options}(G) &= \{ a \in L \mid \exists g \in G \text{ and } s \in \text{Stat} : g{=}\varepsilon{\Rightarrow}s \text{ and } \neg \text{ stable}(s) \text{ and } s{-}a{\rightarrow} \} \\
&= \bigcup_{g \in G} \{ a \in L \mid \exists s \in \text{Stat} : g{=}\varepsilon{\Rightarrow}s \text{ and } \neg \text{ stable}(s) \text{ and } s{-}a{\rightarrow} \} \\
&= \bigcup_{g \in G} \text{Options}(\{g\})
\end{aligned}$$

□

As a consequence of this proposition we have the following results:

- $\text{Compulsory}(G_1 \cup G_2) = \text{Compulsory}(G_1) \cup \text{Compulsory}(G_2)$
- $\text{Options}(G_1 \cup G_2) = \text{Options}(G_1) \cup \text{Options}(G_2)$

Therefore, finding the compulsory or the options sets of a state set G is equivalent to finding these sets for each element of G and then taking the union of the sets found. Now we can give a *recursive* algorithm for computing the compulsory and the options sets.

Algorithm 3.1 (compulsory and options sets)

Proposition 3.1 reduces the problem to finding the compulsory and options sets for singleton sets.
So given a state $s \in S$, let's compute Comp = Compulsory({s}) and Opt = Options({s}).

If stable(s) then Comp $\leftarrow \{ a \in L \mid s\text{–}a\rightarrow \}$ and Opt $\leftarrow \emptyset$
else let $G = \{ s' \in Stat \mid s\text{–}i\rightarrow s' \}$
Comp $\leftarrow$ Compulsory(G)
Opt $\leftarrow \{ a \in L \mid s\text{–}a\rightarrow \} \cup$ Options(G) □

Remark

The generalized CO-OP method does maintain compatibility with the original method. All we did are slight modifications to the definition of the compulsory and options sets so that cycles in the LTS can be handled more easily. The meaning of these sets remains the same. Indeed, definition 3.5, proposition 3.1 and algorithm 3.1 do express the same concepts introduced in chapter 2; a proof of these results will be given later in this chapter.

3.3.4 Finding the orthogonal set

A recursive algorithm for finding the set Orth(A) is given:

Algorithm 3.2 (orthogonal set)

Given a set $A \subseteq \{ X \subseteq L \mid X \neq \emptyset \}$, let's find the set Orth(A) as defined in chapter 2 (definition 2.8).

If $A = \emptyset$ then $\text{Orth}(A) \leftarrow \emptyset$

else suppose $A = \{\{a_1, a_2, \dots a_n\}\} \cup A'$

1. Compute the set $\text{Orth}(A') = \{O_1, O_2, \dots O_p\}$
2. For each set $O_i \in \text{Orth}(A')$ and $a_j \in \{a_1, a_2, \dots a_n\}$, compute $O_{i,j} = O_i \cup \{a_j\}$
3. $\text{Orth}(A) \leftarrow \{ O_{i,j} \mid i = 1, \dots p \text{ and } j = 1, \dots n \}$

3.3.5 Canonical testers and state sets

Definition 3.6 ($T_{can}(B)$)

Let $B = <\text{Stat, Act, Trans}, s_0>$ be a LTS. We define the LTS $T_{can}(B)$ by:

$T_{can}(B) = T(\{s_0\})$ where given G a state set of B, T(G) is defined by:

if $\emptyset \notin \text{Compulsory}(G)$ then

$$T(G) \quad = \quad \textstyle\sum_{V \in \text{Orth(Compulsory}(G))} i; \sum_{a \in V} a; T(G \text{ after } a)$$

$$[] \quad \textstyle\sum_{a \in \text{Options}(G)} a; T(G \text{ after } a)$$

else $\{ \emptyset \in \text{Compulsory}(G) \}$

$$T(G) \quad = \quad i; \text{stop}$$

$$[] \quad \textstyle\sum_{a \in \text{Out}(G)} a; T(G \text{ after } a)$$

where $\text{Out}(G) =_{def} \bigcup_{g \in G} \text{Out}(g)$

Theorem 3.1 (general canonical tester)

$T_{can}(B)$ is a general canonical tester of B. □

To prove theorem 3.1 we will prove that $T_{can}(B)$ is the same LTS generated by the original CO-OP method.

Before proving theorem 3.1, we need to prove the following lemma:

Lemma 3.1

$\text{Compulsory}(\Sigma_j\ i;B_j) = \bigcup_j \text{Compulsory}(B_j)$
$\text{Options}(\Sigma_j\ i;B_j) = \bigcup_j \text{Options}(B_j)$

Proof

Let $B = \Sigma_j\ i;B_j$
Let $\text{Stable}(B) =_{def} \{\ B' \mid B{=}\varepsilon{\Rightarrow}B'{\not-}i{\rightarrow}\ \}$
and $\text{Unstable}(B) =_{def} \{\ B' \mid B{=}\varepsilon{\Rightarrow}B'{-}i{\rightarrow}\ \}$; one can easily see that:
$\text{Stable}(\Sigma_j\ i;B_j) = \bigcup_j \text{Stable}(B_j)$
$\text{Unstable}(\Sigma_j\ i;B_j) = \bigcup_j \text{Unstable}(B_j)$

From definition 2.6 we have:

$$\begin{aligned}
\text{Compulsory}(B) &= \{\ A \subseteq L \mid \exists B' : B{=}\varepsilon{\Rightarrow}B'{\not-}i{\rightarrow} \text{ and } A = \{\ a \in L \mid B'{-}a{\rightarrow}\ \}\ \} \\
&= \{\ A \subseteq L \mid \exists B' \in \text{Stable}(B) \text{ and } A = \{\ a \in L \mid B'{-}a{\rightarrow}\ \}\ \} \\
&= \{\ A \subseteq L \mid \exists B' \in \textstyle\bigcup_j \text{Stable}(B_j) \text{ and } A = \{\ a \in L \mid B'{-}a{\rightarrow}\ \}\ \} \\
&= \textstyle\bigcup_j \{\ A \subseteq L \mid \exists B' \in \text{Stable}(B_j) \text{ and } A = \{\ a \in L \mid B'{-}a{\rightarrow}\ \}\ \} \\
&= \textstyle\bigcup_j \{\ A \subseteq L \mid \exists B' : B_j{=}\varepsilon{\Rightarrow}B'{\not-}i{\rightarrow} \text{ and } A = \{\ a \in L \mid B'{-}a{\rightarrow}\ \}\ \}
\end{aligned}$$

$= \bigcup_j$ Compulsory(B_j)

Options(B) $= \{ a \in L \mid \exists B' : B{=}\varepsilon{\Rightarrow}B'{-}i{\rightarrow}$ and $B'{-}a{\rightarrow} \}$

$= \{ a \in L \mid \exists B' \in$ Unstable(B) and $B'{-}a{\rightarrow} \}$

$= \{ a \in L \mid \exists B' \in \bigcup_j$ Unstable(B_j) and $B'{-}a{\rightarrow} \}$

$= \bigcup_j \{ a \in L \mid \exists B' \in$ Unstable(B_j) and $B'{-}a{\rightarrow} \}$

$= \bigcup_j \{ a \in L \mid \exists B' : B_j{=}\varepsilon{\Rightarrow}B'{-}i{\rightarrow}$ and $B'{-}a{\rightarrow} \}$

$= \bigcup_j$ Options(B_j)

We also need to introduce the following notation.

Notation

Given a state s of B, let B/s $=_{def}$ < Stat, Act, Trans, s >
and given a state set G of B, let B/G $=_{def} \Sigma_{g \in G}$ i; B/g

Proof of theorem 3.1

To avoid confusion, words referring to old definitions (from chapter 2) are underlined. First, we have to prove the following properties:

- Compulsory(G) = <u>Compulsory</u>(B/G) (1)

proof

Compulsory(G) $= \bigcup_{g \in G}$ Compulsory({g})

$= \bigcup_{g \in G}$ <u>Compulsory</u>(B/g) — Definition of compulsory sets

= <u>Compulsory</u>($\Sigma_{g \in G}$ i; B/g) — Lemma 3.1

= <u>Compulsory</u>(B/G) — Definition of B/G

- $\text{Options}(G) = \underline{\text{Options}}(B/G)$ (2)

proof

$\text{Options}(G)$	$= \bigcup_{g \in G} \text{Options}(\{g\})$	
	$= \bigcup_{g \in G} \underline{\text{Options}}(B/g)$	Definition of options sets
	$= \underline{\text{Options}}(\Sigma_{g \in G}\ i;\ B/g)$	Lemma 3.1
	$= \underline{\text{Options}}(B/G)$	Definition of B/G

- $\text{Out}(G) = \underline{\text{Out}}(B/G)$ (3)

proof

$\text{Out}(G)$	$= \bigcup_{g \in G} \text{Out}(g)$	
	$= \bigcup_{g \in G} \underline{\text{Out}}(B/g)$	Definition of Out sets
	$= \underline{\text{Out}}(\Sigma_{g \in G}\ i;\ B/g)$	Definition of Out sets
	$= \underline{\text{Out}}(B/G)$	Definition of B/G

- $B/(G \text{ after } a) = (B/G)\ \underline{\text{after}}\ a$ (4)

proof

$B/(G \text{ after } a)$	$= \Sigma_{g' \in G \text{ after } a}\ i;\ B/g'$	
	$= \Sigma_{g' \in \text{Stat} \mid \exists g \in G\ :\ g = a \rightarrow g'}\ i;\ B/g'$	Definition of G after a
	$= \Sigma_{B' \mid \exists g \in G\ :\ B/g = a \rightarrow B'}\ i; B'$	Definition of B/s
	$= \Sigma_{g \in G}\ \Sigma_{B' \mid B/g = a \rightarrow B'}\ i; B'$	
	$= \Sigma_{g \in G}\ (B/g)\ \underline{\text{after}}\ a$	Definition of B $\underline{\text{after}}$ a
	$= (\Sigma_{g \in G}\ i;\ B/g)\ \underline{\text{after}}\ a$	

= (B/G) <u>after</u> a

- T(G) = <u>T</u>(B/G) (5)

proof

By using (1), (2) and (3) and by making the appropriate substitutions in the definition of T(G) we get:

if Ø ∉ <u>Compulsory</u>(B/G) then
T(G) = $\sum_{V \in Orth(\underline{Compulsory}(B/G))}$ i; $\sum_{a \in V}$ a; T(G after a)
[] $\sum_{a \in \underline{Options}(B/G)}$ a; T(G after a)
else { Ø ∈ <u>Compulsory</u>(B/G) }
T(G) = i; stop
[] $\sum_{a \in \underline{Out}(B/G)}$ a; T(G after a)
While:
if Ø ∉ <u>Compulsory</u>(B/G) then
<u>T</u>(B/G) = $\sum_{V \in Orth(\underline{Compulsory}(B/G))}$ i; $\sum_{a \in V}$ a; <u>T</u>((B/G) <u>after</u> a)
[] $\sum_{a \in \underline{Options}(B/G)}$ a; <u>T</u>((B/G) <u>after</u> a)
else { Ø ∈ <u>Compulsory</u>(B/G) }
<u>T</u>(B/G) = i; stop
[] $\sum_{a \in \underline{Out}(B/G)}$ a; <u>T</u>((B/G) <u>after</u> a)
By using (4) we get:
if Ø ∉ <u>Compulsory</u>(B/G) then
<u>T</u>(B/G) = $\sum_{V \in Orth(\underline{Compulsory}(B/G))}$ i; $\sum_{a \in V}$ a; <u>T</u>(B/(G after a))
[] $\sum_{a \in \underline{Options}(B/G)}$ a; <u>T</u>(B/(G after a))
else { Ø ∈ <u>Compulsory</u>(B/G) }
<u>T</u>(B/G) = i; stop

$[] \quad \sum_{a \in \underline{Out}(B/G)} a;\ \underline{T}(B/(G \text{ after } a))$

We can see that T(G) and $\underline{T}$(B/G) have the same initial behaviour so will T(G after a) and $\underline{T}$(B/(G after a)) and so on ...

So T(G) = $\underline{T}$(B/G)

Finally:

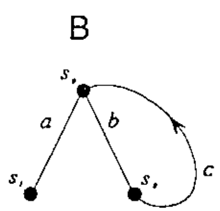
$T_{can}(B)$	$= T(\{s_0\})$	Definition of $T_{can}(B)$
	$= \underline{T}(B/\{s_0\})$	Using (5)
	$= \underline{T}(i;\ B/s_0)$	Definition of B/G
	$= \underline{T}(B/s_0)$	Property of $\underline{T}$
	$= \underline{T}(B)$	□

3.3.6 Building the canonical tester

By looking at the formula representing $T_{can}(B)$, we can see that it may still yield an infinite behaviour expression, even when B is finite. We can iilustrate this by the following example:

Example 3.5

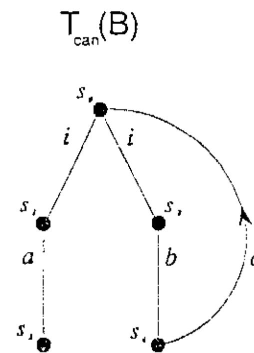
Consider the following LTS:



By applying the formula in definition 3.6 we get an infinite behaviour expression:

$T_{can}(B)$ = $T(\{s_0\})$
= i; a; $T(\{s_1\})$ [] i; b; $T(\{s_2\})$
= i; a; stop [] i; b; c; $T(\{s_0\})$
= i; a; stop [] i; b; c; (i; a; stop [] i; b; c; (i; a; stop ...

In this example it can be easily seen that $T_{can}(B)$ can be represented by a LTS whose set of states is finite. The use of state sets makes it easier to find the corresponding LTS:



We propose an algorithm that can represent in general the behaviour expression of $T_{can}(B)$ by a LTS whose set of states and set of transitions are finite, provided that the set of states and the set of transitions of B are finite.

Algorithm 3.3 (general canonical tester)

Given B = < Stat, Act, Trans, s_0 >, the goal of the algorithm is to find the initial state t_0 and to build the sets Stat′, Act′ and Trans′ for the LTS T = T_{can} (B).
Since Tr(T) = Tr(B), we have Act′ = Act.
Visited $=_{def}$ { (G, t) | G is a state set of B, t ∈ Stat′ and $T_{can}(B/G)$ = T/t }
Tester is the main function of the algorithm: given a state set G of B, it computes

recursively its canonical tester, then it returns its initial state.

<u>Initialization</u>

$Stat' \leftarrow \emptyset$

$Act' \leftarrow Act$

$Trans' \leftarrow \emptyset$

$Visited \leftarrow \emptyset$

<u>Main body of the algorithm</u>

$t_0 \leftarrow$ Tester($\{s_0\}$)

Where in general, for a given state set G of B, Tester(G) is defined as follows:

if $\exists\ t_i \in Stat' : (G, t_i) \in Visited$ then

return t_i

else { G is not visited yet }

Add a new state t_i to Stat'

Add (G, t_i) to Visited

Co $\leftarrow$ Compulsory(G)

Op $\leftarrow$ Options(G)

if $\emptyset \notin$ Co then

Or $\leftarrow$ Orth(Co)

For each set $V \in$ Or do

Add a new state t_j to Stat'

Add transition $< t_i, i, t_j >$ to Trans'

For each action $a \in V$ do

$t_k \leftarrow$ Tester(G after a)

Add transition $< t_j, a, t_k >$ to Trans'

For each action $a \in Op$ do

$t_j \leftarrow$ Tester(G after a)

Add transition $< t_i, a, t_j >$ to Trans′

else { $\emptyset \in$ Co }

Add a new State t_j to Stat′

Add transition $< t_i, i, t_j >$ to Trans′

$O \leftarrow$ Out(G)

For each action $a \in O$ do

$t_k \leftarrow$ Tester(G after a)

Add transition $< t_i, a, t_k >$ to Trans′

return t_i □

A close look at the function Tester reveals that it will never loop endlessly, since the set Visited can't grow infinitely, due to the fact that the number of state sets is finite; indeed, if the number of states of B is n then the number of state sets is 2^n. Note that 2^n is the number of all state sets. In practice the number of *reachable state sets* is much less than 2^n so the number of elements of the set Visited won't grow much larger than n.

3.3.7 Optimizing the canonical tester

In additions to Rules 1, 2 and 3 presented in chapter 2, the following rules can be applied to remove some of the redundant internal actions in the canonical tester.

Rule 4: if $\emptyset \notin$ Compulsory(G) then

if Options(G) = $\emptyset$ and Orth(Compulsory(G)) = {V} then

$$T(G) = \sum_{a \in V} a; T(G \text{ after } a)$$

else

$$T(G) = \sum_{V \in Orth(Compulsory(G))} i; \sum_{a \in V} a; T(G \text{ after } a) \;[]\; \sum_{a \in Options(G)} a; T(G \text{ after } a)$$

The following example illustrates the use of Rule 4.

Example 3.6

Consider the following behaviour expression:

B = i; a; stop [] i; b; stop

Compulsory(B) = {{a}, {b}}

Orth(Compulsory(B)) = {{a, b}}

Options(B) = Ø

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

The internal action in T(B) is redundant.

By applying Rule 4 we get:

T(B) = a; stop [] b; stop

Rule 5: if Ø ∈ Compulsory(G) then

if Out(G) = Ø then

T(G) = stop

else T(G) = i; stop

[] $\sum_{a \in Out(G)}$ a; T(G after a)

The following example illustrates the use of Rule 5.

Example 3.7

Consider the following behaviour expression:

B = stop [] i; stop

Compulsory(B) = {Ø}

Out(B) = Ø

T(B) = i; stop

The internal action in T(B) is redundant.

By applying Rule 5 we get:

T(B) = stop □

At a first glance, Rules 4 and 5 might seem "trivial", but since the process of building the canonical tester is recursive, adding these rules can decrease substantially the size of the canonical tester.

3.4 Chapter summary

In this chapter, we have presented our variation on the CO-OP method. The new CO-OP method is capable of treating infinite LOTOS behaviours. A detailed description of the various algorithm used in implementing the generalized CO-OP method was given.

Chapter 4

The Conformance Relation And Refusal Sets

In this chapter, we will present another application for the notion of state sets introduced in chapter 3. State sets will be used to build *complete failure trees* from LTSs, then various algorithms will be presented to check for interesting relations between complete failure trees. We will be able to prove conformance between LTSs also in the case of infinite behaviours (if they have finite state LTSs).

4.1 Refusal sets and failure trees

Definition 4.1 (refusal set)

Let B be a LTS, s a state of B and σ a string of observable actions. The refusal set $R_\sigma(s)$ is defined as follows:

$R_\sigma(s) = \{ A \subseteq L \mid \exists s' : (s{=}\sigma{\Rightarrow}s' \text{ and } \forall a \in A : s' \not{=}a{\Rightarrow}) \}$ □

The conformance relation can be defined using refusal sets [Led 90]. This definition is equivalent to the one already given in chapter 2 (definition 2.2).

Definition 4.2 (conformance relation and refusal sets)

Let B = < Stat, Act, Trans, s_0 > and I = < Stat′, Act′, Trans′, i_0 > be two LTSs.
I conf S $\Leftrightarrow$ $\forall \sigma \in Tr(S), R_\sigma(i_0) \subseteq R_\sigma(s_0)$.

Problem

Since the set of all strings of observable actions is infinite most of the times, proving that *I conf S* using refusal sets is not feasible. Indeed, in the case of infinite behaviours, we can just prove that *I conf S* is true only up to a given depth (trace length).

Solution strategy : Construct a model for both S and I that can make checking for the *conf* relation more manageable: the model should be deterministic and should preserve the information necessary for checking for the *conf* relation.

Building the model involves two steps:

a) Building an intermediate model: we will call it the *canonical trace equivalent.*
b) Building the final model on top of the canonical trace equivalent: this will be done by adding at each state the refusal set that will be used to check for the *conf* relation. We will present the algorithm in details later in this chapter.

The final model is very similar to the *failure tree* model introduced in [Bri 88], except for the fact that it will handle infinite behaviours with finite state LTSs. Hence, we call the model: the *complete failure tree.*

In [Bri 88] the failure tree model is defined as follows:

Definition 4.3 (failure tree)

A *failure tree* is a 5-tuple < Stat, L, Trans, R, s_0 >, where:

1. Stat is a (countable) non-empty set of states.
2. L is a set of observable actions.
3. Trans = { < s_1, a, s_2 > | s_1, s_2 $\in$ Stat, a $\in$ L } is a set of transitions.
4. R: Stat $\rightarrow 2^L$ is a mapping from the set Stat to the set of subsets of L (power set of L).
5. s_0 is the initial state.

such that:

a) The 4-tuple < Stat, L, Trans, s_0 > is an unordered labelled tree structure with root s_0.
b) $\forall s \in$ Stat, $a \in$ L : $\exists$ at most $s' \in$ Stat | s=a$\Rightarrow s'$.
c) $\forall s \in$ Stat : R(s) $\neq \emptyset$.
d) $\forall s \in$ Stat : if X $\in$ R(s) and Y $\subseteq$ X then Y $\in$ R(s).
e) $\forall s \in$ Stat : if X $\in$ R(s) then X $\cup$ (L - Out(s)) $\in$ R(s). □

In practice, the model defined above is limited to finite behaviours, since the tree will grow only to a certain depth. The model we propose won't be limited to a tree structure; i.e., it can contain cycles. In the following section we will present the first step of building the model.

4.2 Canonical trace equivalent

A *canonical trace equivalent* of a LTS B is a deterministic LTS that has the same set of traces as B. Formally it is defined as follows:

Definition 4.4 (canonical trace equivalent)

Let B be a LTS. A *canonical trace equivalent* of B, denoted by $T_{te}(B)$ is a LTS that satisfies:

(i) $Tr(T_{te}(B)) = Tr(B)$; and

(ii) $T_{te}(B)$ is *deterministic*. We define this to mean that:

if $s_1 = \sigma \Rightarrow s_2$ and $s_1 = \sigma \Rightarrow s_3$ then $s_2 = s_3$

There are other definitions of the notion of *determinism* in the literature. □

We can easily see that : (ii) implies that $\forall s : s \not\xrightarrow{i}$

4.2.1 Building the canonical trace equivalent

We propose a systematic algorithm for building the canonical trace equivalent for any given finite state LTS B. The algorithm is recursive and uses the notion of state sets.

Theorem 3.1

Let B = < Stat, Act, Trans, s_0 >.

Consider the LTS $T_{te}(B)$ defined by the following:

$T_{te}(B) = T(\{s_0\})$ where given a state set G of B, T(G) is defined by:

$T(G) = \sum_{a \in Out(G)} a; T(G \text{ after } a)$

$T_{te}(B)$ is a canonical trace equivalent of B. □

The proof of the theorem follows directly from the definitions of *Out(G)* and *G after a*.

The LTS B may contain cycles of internal actions. The computation of the set *Out(s)* may then loop endlessly. For the purpose of completeness and to avoid the problem of looping

endlessly, we propose an algorithm that computes the set *Out(G)*.
Since $Out(G) =_{def} \bigcup_{g \in G} Out(g)$, this reduces the problem to computing the set *Out(s)* for a given state *s*.

Algorithm 4.1 (Out set)

Let B = < Stat, Act, Trans, s_0 > be a LTS and s a state from Stat. The algorithm computes the set O = Out(s).
Visited is a set of sates that is used to detect cycles of internal actions in B. It is initially set to empty.

<u>Initialization</u>

Visited ← Ø

O ← Ø

<u>Main body of the algorithm</u>

Add s to Visited

O ← O ∪ { a ∈ L | s–a→ }

For each s′ such that s–i→s′ do

if s′ ∉ Visited then

O ← O ∪ Out(s′) □

We then propose the algorithm that can represent in general the behaviour expression of $T_{te}(B)$ by a LTS whose set of states and transitions are finite.

Algorithm 4.2 (canonical trace equivalent)

Given B = < Stat, Act, Trans, s_0 >, the goal of the algorithm is to find the initial state t_0

and to build the sets Stat′, Act′ and Trans′ for the LTS $T = T_{te}(B)$.
Since Tr(T) = Tr(B), we have Act′ = Act - {i} = L ($i \notin$ Act′ since T is deterministic).
Visited $=_{def}$ { (G, t) | G is a state set of B, $t \in$ Stat′ and $T_{te}(B/G) = T/t$ }
Tracer is the main function of the algorithm: given a state set G of B, it computes recursively the canonical trace equivalent of B/G, then it returns its initial state.

Initialization

Stat′ ← Ø

Act′ ← Act - {i}

Trans′ ← Ø

Visited ← Ø

Main body of the algorithm

t_0 ← Tracer({s_0})

Where in general, for a given state set G of B, Tracer(G) is defined as follows:

if $\exists\ t_i \in$ Stat′ : (G, t_i) $\in$ Visited then

return t_i

else { G is not visited yet }

Add a new state t_i to Stat′

Add (G, t_i) to Visited

O ← Out(G)

For each action $a \in$ O do

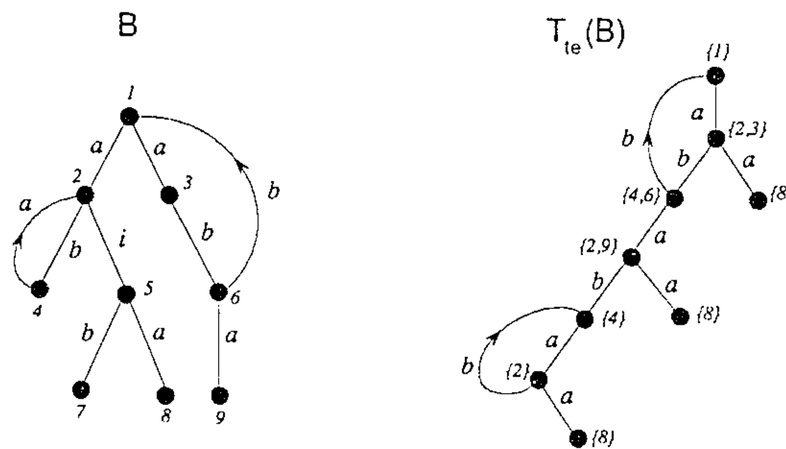
t_j ← Tracer(G after a)

Add transition $< t_i, a, t_j >$ to Trans′

return t_i

Example 4.1

In the following example $T_{te}(B)$ was constructed using algorithm 4.2.



$T_{te}(B)$ has the same traces as B but it is deterministic.

Remark

In general the number of states in $T_{te}(B)$ is equal to the number of *possible state sets* in B where:

A state set G is possible iff $G \neq \emptyset$ and $\exists \sigma \in L^* : G = \{s_0\}$ after σ. □

4.2.2 Trace inclusion

The canonical trace equivalent model can be used to check for trace inclusion between LTSs: given B and B′ two LTSs, do we have $Tr(B) \subseteq Tr(B')$?

We propose an algorithm that will answer that question for any pair of finite state LTSs. Since $Tr(B) = Tr(T_{te}(B))$ and $Tr(B') = Tr(T_{te}(B'))$ checking if $Tr(B) \subseteq Tr(B')$ is equivalent

to checking if $Tr(T_{te}(B)) \subseteq Tr(T_{te}(B'))$. This reduces the problem to checking trace inclusion for deterministic LTSs.

Algorithm 4.3 (trace inclusion)

Let B and B′ be two LTSs.
The first step of the algorithm consists of building $T = T_{te}(B)$ and $T' = T_{te}(B')$.
Suppose $T = < Stat, Act, Trans, s_0 >$ and $T' = < Stat', Act', Trans', s'_0 >$.
To each state s of T we associate a set of states of T′ : Checked(s) where:
$\forall s \in Stat : Checked(s) = \{s' \in Stat' \mid Tr(s) \subseteq Tr(s')\}$

Initialization

$\forall s \in Stat$: let $Checked(s) = \emptyset$

Main body of the algorithm

$Tr(T) \subseteq Tr(T') \Leftrightarrow Incl(s_0, s'_0)$

Where in general, given $s \in Stat$ and $s' \in Stat'$: Incl(s, s′) is TRUE iff $Tr(s) \subseteq Tr(s')$.
The boolean function Incl(s, s′) is defined recursively as follows:

```
if s′ ∈ Checked(s) then
        return TRUE
else { s′ ∉ Checked(s) }
        if Out(s) ⊆ Out(s′) then
                Add s′ to Checked(s)
                if Out (s) = Ø then
                        return TRUE
                else    return ∧ a ∈ Out(s) and s–a→t and s′–a→t′ Incl(t, t′)
        else    return FALSE
```

4.3 Complete failure trees

First, we propose an extension of the notion of refusal sets to state sets.

Definition 4.5 (refusal set of a state set)

Let B be a LTS, G a state set of B and σ a string of observable actions.

$R_\sigma(G) =_{def} \bigcup_{g \in G} R_\sigma(g)$ □

As we have seen in chapter 3, state sets are used to represent the behaviour expression $B/G = \Sigma_{g \in G}$ i; B/g. Since refusal sets have the following property, it is understood why we have chosen definition 4.5 for state sets.

Proposition 4.1

$\underline{R}_\sigma(\Sigma_j\ i;B_j) = \bigcup_j \underline{R}_\sigma(B_j)$ where:

$\underline{R}_\sigma(B/s) =_{def} R_\sigma(s)$

Proof

$\underline{R}_\sigma(\Sigma_j\ i;B_j) = \{\ A \subseteq L \mid \exists B' : (\Sigma_j\ i;B_j =\sigma\Rightarrow B' \text{ and } \forall a \in A : B' \neq a \Rightarrow)\ \}$

Since $\Sigma_j\ i;B_j =\sigma\Rightarrow B' \Leftrightarrow \bigvee_j B_j=\sigma\Rightarrow B'$ we get:

$\underline{R}_\sigma(\Sigma_j\ i;B_j) = \bigcup_j \{\ A \subseteq L \mid \exists B' : (B_j =\sigma\Rightarrow B' \text{ and } \forall a \in A : B' \neq a \Rightarrow)\ \} = \bigcup_j \underline{R}_\sigma(B_j)$.

Notation

$Ref(s) =_{def} R_\varepsilon(s)$

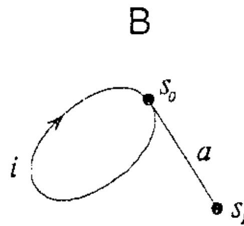
$Ref(G) =_{def} R_\varepsilon(G)$ □

Remark

For the computation of the Refusal sets, we assume that the LTS in question does not contain cycles of internal actions. This assumption is not considered as a limitation since such cycles in a given specification can be judged as design faults. Cycles of internal actions can be easily detected in a LTS. The presence of cycles of internal actions in a LTS makes the concept of refusal sets ambiguous; this can be illustrated by the following example:

Example 4.2

Consider the following LTS:



By looking at B, we can see that it is not clear whether B may refuse action *a* or not. The answer to this question depends on a fairness assumption.

Let's try to compute the set $Ref(s_0) = R_\varepsilon(s_0)$:

$R_\varepsilon(s_0) =_{def} \{ A \subseteq L \mid \exists s' : (s_0 = \varepsilon \Rightarrow s' \text{ and } \forall a \in A : s' \not= a \Rightarrow) \}$

$= \{ A \subseteq L \mid \forall a \in A : s_0 \not= a \Rightarrow \}$ since B has one state $s' = s_0$ such that $s_0 = \varepsilon \Rightarrow s'$

$\{a\} \notin R_\varepsilon(s_0)$ since $s_0 = a \Rightarrow$

$R_\varepsilon(s_0) \neq \emptyset$ since $\varepsilon \in Tr(s_0)$

So $R_\varepsilon(s_0) = \{\emptyset\}$ which says that B will never refuse action *a*; but one can argue that B may refuse action *a* by engaging in an infinite loop of internal actions. □

We propose an algorithm for computing Ref(G).

Algorithm 4.4 (refusal set)

Let B = < Stat, Act, Trans, s_0 > be a LTS and G a state set of B.
We can easily see that $\text{Ref}(G) = \bigcup_{g \in G} \text{Ref}(g)$. This reduces the problem to finding Ref(s) for a given state *s* of B. The set *Visited* is used to detect cycles of internal actions.
The algorithm uses rule *R 4.7 b)* for computing the refusals of compound processes with the choice operator introduced in [Gal 89] and [GLO 91]:

$$\text{refusals}(i;\ B_1\ [\,]\ B_2) = \text{refusals}(B_1)$$

Let's compute R = Ref(s):

<u>Initialization</u>

Visited ← Ø

<u>Main body of the algorithm</u>

if stable(s) then R ← { L - Out(s) }

else add s to Visited

$R \leftarrow \bigcup_{s' \in \text{Stat} \mid s-i\rightarrow s' \text{ and } s' \notin \text{Visited}} \text{Ref}(s')$

Remark

Notice that that the algorithm keeps only the maximal sets in Ref(s):
$\forall\ A \in \text{Ref}(s) : \nexists\ B \in \text{Ref}(s) \mid B \subseteq A$
The same procedure can be applied when building $\text{Ref}(G) = \bigcup_{g \in G} \text{Ref}(g)$: we remove the the redundant sets from Ref(G) so that: $\forall\ A \in \text{Ref}(G) : \nexists\ B \in \text{Ref}(G) \mid B \subseteq A$. □

Formally the complete failure tree of a LTS is defined as follows:

Definition 4.6 (complete failure tree)

Let B = < Stat, Act, Trans, s_0 > be a LTS.
Consider the 5-tuple $F_c(B)$ = < Stat′, L′, Trans′, R, t_0 > where:

1. Stat′ is a (countable) non-empty set of states.
2. L′ is a set of observable actions.
3. Trans′ = { < s_1, a, s_2 > | s_1, s_2 ∈ Stat′, a ∈ L′ } is a set of transitions.
4. R: Stat′ → $2^{L'}$ is a mapping from the set Stat′ to the power set of L′.
5. t_0 is the initial state.

$F_c(B)$ is said to be a complete failure tree of B iff:

(i) The 4-tuple < Stat′, L′, Trans′, t_0 > is a canonical trace equivalent of B.
(ii) ∀t ∈ Stat′ : if t_0–σ→t and G = {s_0} after σ then R(t) = Ref(G). □

We propose an algorithm for building a complete failure tree for a given finite state LTS.

Algorithm 4.5 (complete failure tree)

Given B = < Stat, Act, Trans, s_0 >, the goal of the algorithm is to find the initial state t_0 and to build the sets Stat′, Act′ and Trans′ and the mapping R for F = $F_c(B)$.
Since Tr(F) = Tr(B), we have Act′ = Act - {i} = L (i ∉ Act′ due to (ii) of definition 4.6).
Visited $=_{def}$ { (G, t) | G is a state set of B, t ∈ Stat′ and $F_c(B/G)$ = F/t }
Failure is the main function of the algorithm: given a state set G of B, it computes recursively its complete failure tree, then it returns its initial state.

<u>Initialization</u>

$Stat' \leftarrow \emptyset$

$Act' \leftarrow Act - \{i\}$

$Trans' \leftarrow \emptyset$

$Visited \leftarrow \emptyset$

<u>Main body of the algorithm</u>

$t_0 \leftarrow$ Failure($\{s_0\}$)

Where in general, for a given state set G of B, Failure(G) is defined as follows:

if $\exists\ t_i \in Stat' : (G, t_i) \in Visited$ then

return t_i

else { G is not visited yet }

Add a new state t_i to Stat′

Add (G, t_i) to Visited

$R(t_i) \leftarrow Ref(G)$

$O \leftarrow Out(G)$

For each action $a \in O$ do

$t_j \leftarrow$ Failure(G after a)

Add transition $< t_i, a, t_j >$ to Trans′

return t_i □

Note that $T_{tc}(B)$ is generated as part of the algorithm for generating $F_c(B)$.

4.4 Proving conformance using complete failure trees

In this section we will present an algorithm for proving conformance between LTSs in the following special case:

For reasons already mentioned, the algorithm imposes the following restrictions on $B = <\text{Stat, Act, Trans}, s_0>$ and $B' = <\text{Stat}', \text{Act}', \text{Trans}', s'_0>$:

a) The sets Stat, Act, Trans, Stat′, Act′ and Trans′ must be finite.

b) B and B′ do not contain cycles of internal actions.

This restriction does not prevent B and B′ from describing a wide set of infinite behaviours, since B and B′ can contain cycles of actions.

Remark

In the algorithm we will assume that Act = Act′. If that is not the case we can consider the set Act ∪ Act′ as the set of actions for both B and B′.

Algorithm 4.6 (proving conformance)

Let B and B′ be two LTSs.

The first step of the algorithm consists of building $F = F_c(B)$ and $F' = F_c(B')$.

Suppose $F = <\text{Stat, Act, Trans}, R, s_0>$ and $F' = <\text{Stat}', \text{Act}', \text{Trans}', R', s'_0>$.

To each state s of F we associate a set of states of F′ : $\text{Checked}(s) \subseteq \text{Stat}'$.

Initialization

$\forall s \in \text{Stat} : \text{let Checked}(s) = \emptyset$

Main body of the algorithm

$B \text{ conf } B' \Leftrightarrow \text{Conf}(s_0, s'_0)$

Where in general, given $s \in \text{Stat}$ and $s' \in \text{Stat}'$, the boolean function Conf(s, s′) is defined recursively as follows:

if $s' \in$ Checked(s) then

    return TRUE

else { $s' \notin$ Checked(s) }

    if $\forall A \in R(s) : \exists A' \in R'(s') \mid A \subseteq A'$ then

        Add s' to Checked(s)

        if $Out(s) \cap Out(s') = \emptyset$ then

            return TRUE

        else   return $\bigwedge_{a \in Out(s) \cap Out(s') \text{ and } s-a\rightarrow t \text{ and } s'-a\rightarrow t'} Conf(t, t')$

    else   return FALSE

4.5 Limitations of the approach

Proving trace inclusion or conformance assumes that we have the models (LTS or symbolic execution tree) of both the specification and the IUT. In practice that assumption is not always true, indeed, the IUT can be written in any language (C, Pascal, Prolog ...) and obtaining a model from a source code written in any of these languages is not always feasible.

4.6 Chapter summary

In this chapter we presented another application for the notion of state sets. We used state sets to build *complete failure trees* from LTSs, then we presented various algorithms to check for interesting relations between complete failure trees. We proved that conformance between LTSs can be proven, also in the case of infinite Basic LOTOS behaviours.

Chapter 5

Canonical Testers Of Full LOTOS Specifications

There is a resemblance between full LOTOS and basic LOTOS testing theory, with some differences that may cause the CO-OP method to fail to produce canonical testers for full LOTOS specifications. Instead of modifying the CO-OP method so it handles full LOTOS behaviours, we have borrowed ideas from the *Relabel method* introduced in [Doo 91] for deriving general canonical testers for Full LOTOS specifications. This method modifies the behaviour itself to a testing equivalent behaviour so that applying the CO-OP method to the resulting behaviour produces a canonical tester. A critique of the Relabel method is given.

5.1 Parameterized labelled transition systems

Like the semantics of Basic LOTOS, the semantics of Full LOTOS can also be defined in terms of labelled transition systems. However, simple Full LOTOS behaviour expressions may be defined with an initial behaviour that is infinitely branching. In this case the algorithm implementing the CO-OP method could not establish the initial behaviour of the tester in a finite time. This problem can be solved by using a new type

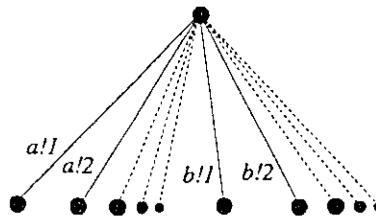
of LTSs that can express the *infinite branching* in a finite way: these are the *parameterized labelled transition systems* (pLTS) first introduced in [Wez 90].

Example 5.1

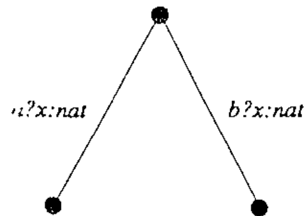
Consider the following Full LOTOS behaviour expression:

B = a?x:nat; stop [] b?x:nat; stop

Representing B by a LTS gives the following infinitely branching structure:



With a pLTS we get:



As we see, unlike the LTS, the pLTS is finitely branching. □

LOTOS specifications define processes that can progress by performing internal transitions or by synchronizing with the environment on external transitions. For pLTSs, external transitions contain a combination of a *gate name* and a series of *experiment offers*. The gate name specifies the gate at which the transition takes place; the experiment offer defines the type of data exchange that occurs (see example 5.1). Transitions for pLTSs are called *parameterized transitions*.

Unfortunately, applying the CO-OP method to pLTS without doing any adaptation of the algorithm does not always produce canonical testers. In the following sectior we will give several counterexamples.

5.2 Applying the CO-OP method to pLTSs

The CO-OP method fails to produce canonical testers for pLTSs, in two situations:

(i) The pLTS contains *overlapping transitions*.

(ii) Some internal events are associated with predicates.

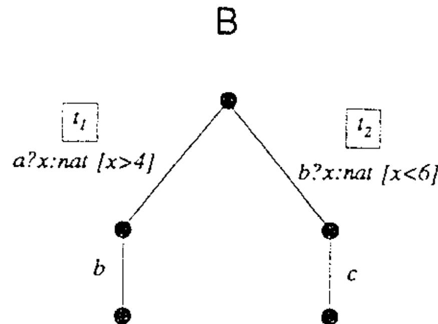
5.2.1 Overlapping transitions

In a LTS, transitions are either equal or disjoint, since each transition describes exactly one action that can be taken. In a pLTS, parameterized transitions can also *overlap* [Doo 91].

The following example illustrates the situation of overlapping transitions, shows the problem after applying the CO-OP method and gives a hint to a general solution for the problem.

Example 5.2

Consider the following pLTS:



We can see that transitions t_1 and t_2 are neither equal nor disjoint. That's the first difference between LTSs and pLTSs.

At this point we need to introduce the following notation:

Notation

The symbol "†" is borrowed from [Doo 91]. It is used to describe the tester offering an event *a!x*, where the tester chooses the value of the experiment offer *x*.
a!x could not be used since in LOTOS a value offer *!x* must always be preceded by a variable declaration ?x □

Applying the CO-OP as we have seen so far to derive a canonical tester of the pLTS B of example 5.2, we get:

T(B) = i; a†x:nat [x>4]; b; stop [] i; a†x:nat [x<6]; c; stop

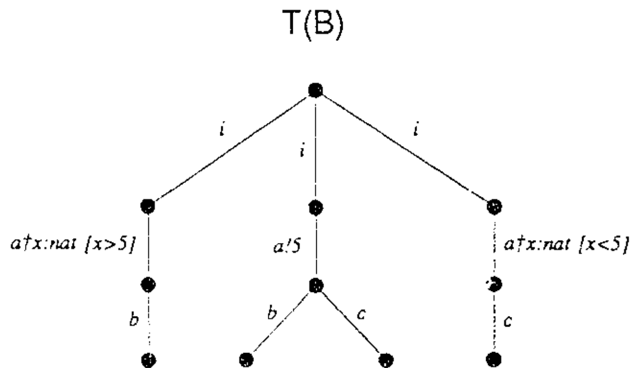
We can easily see that T(B) is not a canonical tester of B, since T(B) may deadlock with B before reaching a terminal state. Consider for example the following test case extracted from T(B):

T = a!5; b; stop

T may deadlock with B since the value *5* satisfies both conditions *x>4* and *x<6*. So the test case must offer a choice of both actions *b* and *c* after *a!5*. One way of solving this problem is to find a pLTS B′ *testing equivalent* to B and that does not have overlapping transitions. In our example that pLTS can be the following:

B′ = a?x:nat [x>5]; b; stop [] a!5; b; stop [] a!5; c; stop [] a?x:nat [x<5]; c; stop

The actual canonical tester of B is:



Finding an *efficient* algorithm that can build a pLTS B′ testing equivalent to B that does not have overlapping transitions requires a kind of *Theorem proving*. However, a "brute force" algorithm can be used for that purpose, assuming that all parameterized

transitions with name-sort identical[1] labels may overlap. The idea of the algorithm is to add transitions with predicate combinations, in such a way that the resulting transitions are mutually exclusive. The algorithm is based on the relabel method II from [Doo 91].

Algorithm 5.1 (overlapping transitions)

Suppose t_1, t_2, ... t_n are name-sort identical transitions with labels belonging to the same *Out* set of a given state of a pLTS B. B′ is obtained by substituting each transition t_i = < s, g?x:s [P_i], s′ > by the following set of transitions:

$$\mathrm{Sub}(t_i) = \{\ < s, g?x{:}s\ [P_i \wedge Q_r], s' > \text{ such that: } Q_r \in \{ \textstyle\bigwedge_{1\le k\le n \text{ and } k\ne i} b(P_k) \mid b(P_k) \in \{P_k, \neg P_k\}\ \}\ \}$$ □

Applying algorithm 5.1 to the pLTS B in example 5.2 gives:

B″	=	a?x:nat [(x>4) and ¬(x<6)]; b; stop
	[]	a?x:nat [(x>4) and (x<6)]; b; stop
	[]	a?x:nat [(x<6) and ¬(x>4)]; c; stop
	[]	a?x:nat [(x<6) and (x>4)]; c; stop

And we have: B″≈ B and B″ ≈ B′.

Remark

Suppose we have *n* overlapping transitions, then each transition t_i will be replaced by 2^{n-1} (number of possible Q_r's) new transitions. This gives an idea about the size of the

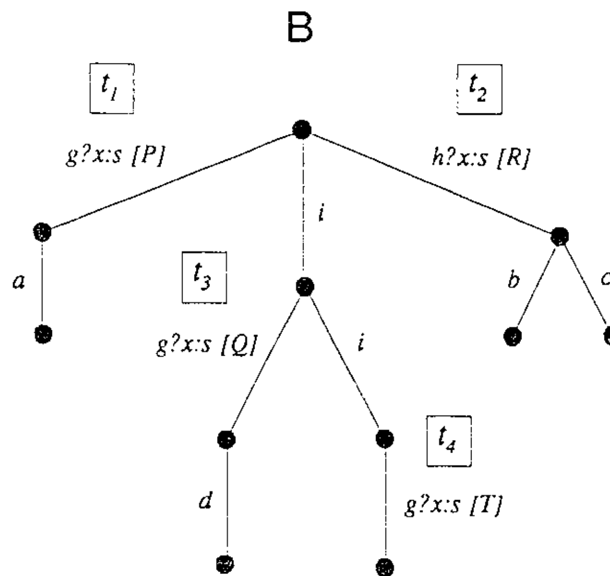
---

[1] This term was first introduced in [Doo 91]. Two labels $g_1?x{:}s_x\ [P]$ and $g_2?y{:}s_y\ [Q]$ are said to be name-sort identical if (i) $g_1 = g_2$ and (ii) $s_x = s_y$

resulting pLTS after applying algorithm 5.1 for eliminating the overlapping transitions.

Example 5.3

Consider the following pLTS:



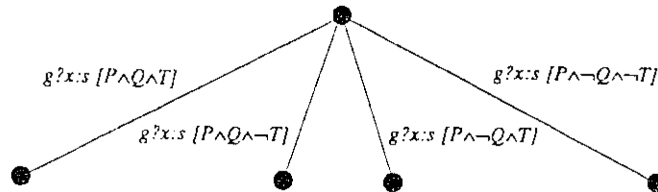
We assume that transitions t_1, t_3 and t_4 may overlap since they are name-sort identical and belong to the same *Out* set of the initial state of the pLTS B.

Transition t_2 will never overlap with transitions t_1, t_3 and t_4 since the gate name associated with its label is different from the one in t_1, t_3 and t_4.

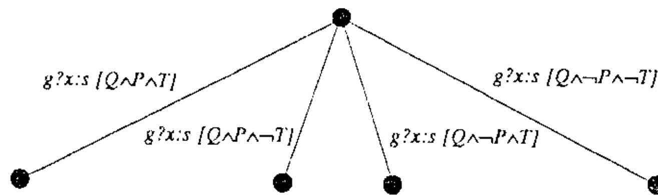
Each transition of $\{t_1, t_3, t_4\}$ will be replaced by $4 = 2^{3-1}$ new transitions.

Applying algorithm 5.1 to eliminate the possibility of overlapping transitions we get:

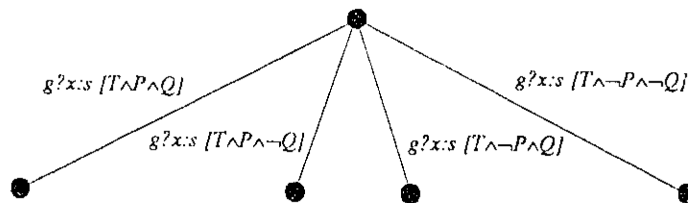
Transition t_1 is replaced by the following transitions:



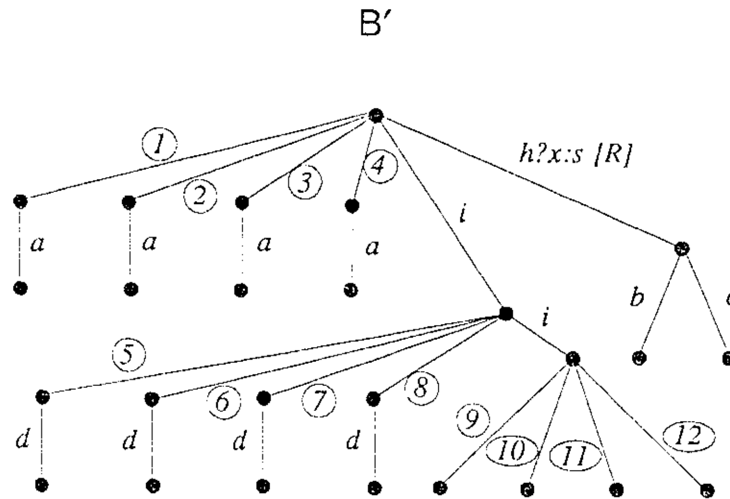
Transition t_3 is replaced by the following transitions:



Transition t_4 is replaced by the following transitions:



We get the following testing equivalent pLTS:



where:

(1)	g?x:s [P∧Q∧T]	(7)	g?x:s [Q∧¬P∧T]
(2)	g?x:s [P∧Q∧¬T]	(8)	g?x:s [Q∧¬P∧¬T]
(3)	g?x:s [P∧¬Q∧T]	(9)	g?x:s [T∧P∧Q]
(4)	g?x:s [P∧¬Q∧¬T]	(10)	g?x:s [Q∧P∧¬T]
(5)	g?x:s [Q∧P∧T]	(11)	g?x:s [Q∧¬P∧T]
(6)	g?x:s [Q∧P∧¬T]	(12)	g?x:s [Q∧¬P∧¬T]

We can easily see that the transitions listed above will never overlap. Therefore, the CO-OP method can be applied to B′ and will generate a general canonical tester of B.

5.2.2 Deadlocking transitions

In addition to the overlapping transitions problem, another problem is encountered while applying the CO-OP method to pLTSs. As described in [Doo 91], the problem of deadlocking parameterized transitions is caused by the presence of predicates in parameterized transitions and occurs in the following two cases:

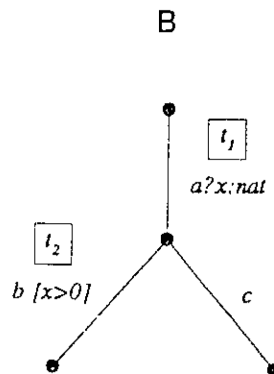
(i) The predicate depends on an already bound variable and is not true.

(ii) There is an experiment offer, but no value fulfils the predicate (which may depend on already bound variables).

The following example illustrates case (i).

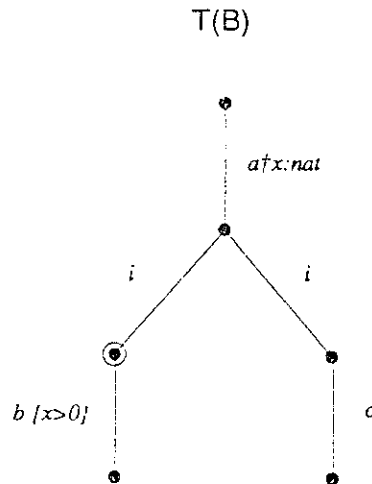
Example 5.3

Consider the following pLTS:



We can see that transition t_2 may yield no transition if $x = 0$: it is a deadlocking transition.

Applying the CO-OP method to B gives the following tester:



One can say that T(B) is not a canonical tester of B, since it may deadlock with B before reaching a terminal state. T(B) deadlocks with B if it chooses $x=0$ and then moves to the state marked by a circle.

An example illustrating case (ii) could be obtained by replacing action *b* $[x>0]$ in example 5.3 with action *b?y.nat* $[(y>0) \wedge (x>0)]$. If $x = 0$ there will be no experiment offer at gate *b* since the condition $[(y>0) \wedge (0>0)]$ is never satisfied. □

At this point let's recall the definition of a general canonical tester from chapter 2:

> A general canonical tester T(S) of a specification S is a process that is (i) capable of exploring all and only traces in S, and (ii) must satisfy the following property: for any process P, P conforms to S iff every deadlock between P and T(S) can be explained by the tester reaching a terminal state.

Property (ii) was formally stated as follows (from definition 2.5):

For all processes P, P conf S iff
for all $\sigma \in L^*$, $P \| T(S) = \sigma \Rightarrow B$ and $B \approx stop$ implies $T(S) = \sigma \Rightarrow B'$ and $\underline{B' \approx stop}$

We can easily see that a terminal state is not interpreted as a state with no outcoming transitions, but as a state in which the tester behaves like *stop*. Using this interpretation of a terminal state, deadlocking transitions no longer cause a problem and testers generated by the CO-OP method from specifications containing deadlocking transitions are general canonical testers.

The *relabel method III* introduced in [Doo 91] includes a step for treating deadlocking transitions which we believe is unnecessary for these reasons. The solution proposed by the method is also very *costly* and cannot be fully automated since at some point it needs to compute a set (DL) and that computation needs some kind of *theorem proving* or "human interventions". This is an extract from the relabel method III:

6) For every deadlock possibility in S:

$$\sigma \in Tr_p(S) : S = \sigma \Rightarrow -i \rightarrow \Sigma \; \{ \; a?x_i{:}s_i[P_i(x_i,\overline{y})]; \; B_i(x_i,\overline{y}) \mid 1 \leq i \leq n \; \}$$

add the deadlock branch $i\ [\overline{y} \in DL];\ stop$ to every state B in T_p with $T_p = \sigma \rightarrow B$
where $DL = \{ \; \overline{y} \mid \forall \; \overline{x} \in \Pi_{1 \leq i \leq n} \; s_i : \wedge_{1 \leq i \leq n} \neg \; P_i(x_i,\overline{y}) \; \}$

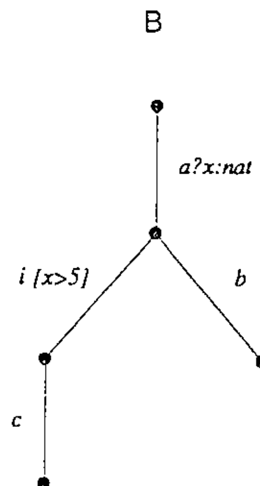
Computing the set DL may require solving a set of mathematical equations that can be very complex.

5.2.3 Internal events with predicates

The presence of predicates associated with internal events causes the CO-OP method to fail. The presence of internal events with predicates in pLTSs is due to LOTOS *guards*; e.g., the following guarded behaviour expression: *[x>1]→i;a;stop* will yield the transition labelled *i [x>1]* containing an internal event with a predicate. As stated in [Doo 91], difficulties arise in computing the Compulsory and the Options sets since external events may be compulsory or optional depending on the value of the predicate associated with the internal events.

Example 5.4

Consider the following pLTS:

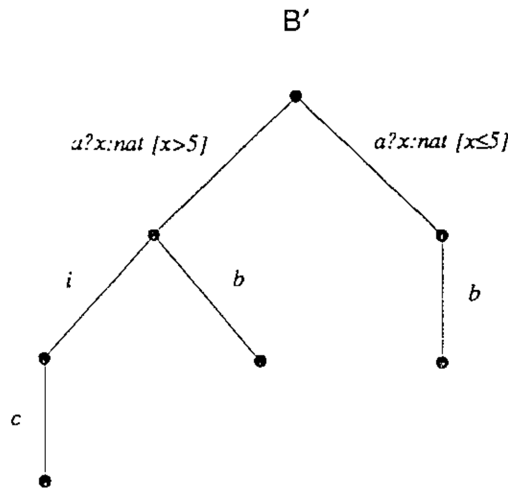


Action b is compulsory or optional depending on the value of the variable x.

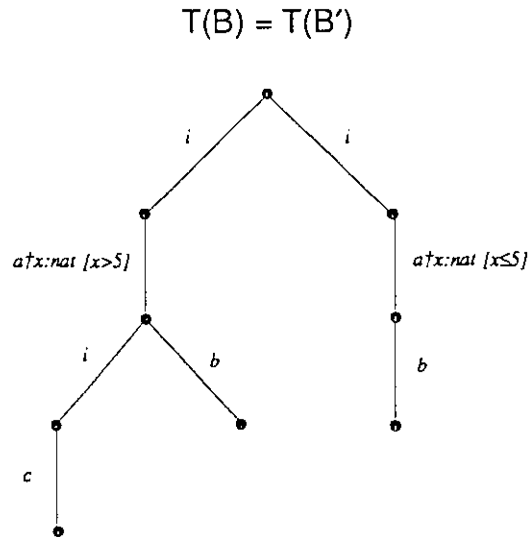
The solution strategy for solving this problem is similar to the one introduced for solving the problem of overlapping transitions: construct a testing equivalent pLTS to B that does

not contain internal events associated with predicates; then apply the CO-OP method to that pLTS to generate a general canonical tester.

B is testing equivalent to the following pLTS:



Applying the CO-OP method on B′, generates a general canonical tester of B.



Finding B′ in general, here again, requires the splitting of many transitions which may result in an explosion of the size of the pLTS. □

Our approach in solving the problem is different from the approach proposed in [Doo 91] which is too lengthy to introduce in this chapter.

Given a pLTS B that contains internal events with predicates, the following algorithm builds a testing equivalent pLTS B′ that does not contain internal events with predicates.

Algorithm 5.2 (internal events with predicates)

The algorithm uses the following property of LOTOS behaviours:

Consider $B = \ldots\ a;\ (b_1;\ B_1\ []\ b_2;\ B_2\ []\ \ldots\ b_k\ [P(x)];\ B_k\ []\ \ldots\ b_n;\ B_n)$ then we have:

$$
\begin{aligned}
B \approx\ & \ldots\ a\ [P(x) \vee \neg P(x)];\ (b_1;\ B_1\ []\ b_2;\ B_2\ []\ \ldots\ b_k\ [P(x)];\ B_k\ []\ \ldots\ b_n;\ B_n) \\
\approx\ & \ldots\ (\ \ a\ [P(x)];\ (b_1;\ B_1\ []\ b_2;\ B_2\ []\ \ldots\ b_k\ [P(x)];\ B_k\ []\ \ldots\ b_n;\ B_n) \\
& \qquad []\ \ a\ [\neg P(x)];\ (b_1;\ B_1\ []\ b_2;\ B_2\ []\ \ldots\ b_k\ [P(x)];\ B_k\ []\ \ldots\ b_n;\ B_n) \\
& \qquad) \\
\approx\ & \ldots\ (\ \ a\ [P(x)];\ (b_1;\ B_1\ []\ b_2;\ B_2\ []\ \ldots\ b_k;\ B_k\ []\ \ldots\ b_n;\ B_n) \\
& \qquad []\ \ a\ [\neg P(x)];\ (b_1;\ B_1\ []\ b_2;\ B_2\ []\ \ldots\ b_{k-1};\ B_{k-1}\ []\ b_{k+1};\ B_{k+1}\ []\ \ldots\ b_n;\ B_n) \\
& \qquad)
\end{aligned}
$$

Note that the predicate associated with b_k was pushed upward to action a. This same procedure can be used if $b_k = i$ to remove the predicate $P(x)$ associated with it. If the action a is an internal action the same process can be used to remove the predicates $P(x)$ and $\neg P(x)$. At some point of applying this recursive process we may reach the following case:

$B = i\ [P(x)];\ B_0\ []\ b_1;\ B_1\ []\ ...\ b_n;\ B_n$ where B is not prefixed by any action.

Here the predicate *P(x)* can no longer be pushed upward. To solve this problem, we use the following property:

$$\begin{aligned} B &\approx i;\ (i\ [P(x)];\ B_0\ []\ b_1;\ B_1\ []\ ...\ b_n;\ B_n) \\ &\approx \quad i\ [P(x)];\ (i;\ B_0\ []\ b_1;\ B_1\ []\ ...\ b_n;\ B_n) \\ &\quad []\ \ i\ [\neg P(x)];\ (b_1;\ B_1\ []\ ...\ b_n;\ B_n) \end{aligned}$$

We can see that we are still left with internal actions associated with predicates. But these cause no difficulty in determining which actions are compulsory or options. Indeed if we look carefully at the resulting behaviour expression we have in that case:

$$\begin{aligned} T(B) &= \quad i\ [P(x)];\ T(i;\ B_0\ []\ b_1;\ B_1\ []\ ...\ b_n;\ B_n) \\ &\quad []\ \ i\ [\neg P(x)];\ T(b_1;\ B_1\ []\ ...\ b_n;\ B_n) \end{aligned}$$

5.3 Chapter summary

In this chapter, we have presented problems involved when applying the CO-OP method to Full LOTOS behaviours for deriving general canonical testers. We proposed solutions for solving these problems. Most of the ideas presented in this chapter were taken from the relabel method introduced in [Doo 91]. A critique of the method was given. The major problem with this method is that canonical testers produced by the method tend to be very large due to the splitting of transitions. This limits the use of the method to small examples. The algorithms described above were partially implemented in LOTEST. At the moment, we assume that the pLTS has a tree structure.

Chapter 6

Implementation Issues

6.1 The Ottawa University LOTOS tool kit

The LOTOS group of the University of Ottawa has developed a prototypal environment for the design and validation of concurrent systems based on the integration between the following tools:

- **ISLA** [HH 89], an interpreter that helps simulating the specifications and discovering design errors by giving the designer the ability to monitor and trace execution sequences.

- **SELA** [Ash 92], a tool that generates the behaviour tree of a specification (or process) symbolically, that is, without the use of actual values.

- **LMC** [Ghr 92], a model checker for LOTOS that is able to formally verify whether certain properties representing the initial requirements expressed in CTL, hold on symbolic behaviour trees generated by SELA from LOTOS specifications.

- **LOTEST**, the implementation of the various ideas found in this thesis, is 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. A detailed description of LOTEST will be given in Appendix A.

The validation environment that we propose supports different functionalities to analyze LOTOS specifications. The main functionalities are:

- **Interactive simulation**: The designer is able to monitor the specification and its behaviour through a step-by-step execution. This can be done by the help of simulators such as HIPPO [VE 89] and ISLA. ISLA provides a wide range of services [GHL 88]. 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. 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 [HH 89]. It is possible to go back to previous points in the execution and explore other alternatives.

- **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 tries 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 [GL 89, QFP 88]. A similar system to SELA is LOLA (LOTOS Laboratory) [QFP 88] which is a transformation tool for LOTOS. It provides several functionalities such as symbolic execution of specifications using various types of expansions.

- **Model checking**: 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 (in our case, the symbolic execution tree generated by SELA). This method is referred to as *model checking*. *Temporal logic* is used to express the correctness specifications while the model checking technique attempts to prove that the finite state system meets these correctness specifications.

- **Test case derivation**: Canonical testers are derived from execution trees using LOTEST's implementation of the CO-OP method. Test cases can be then extracted from the resulting canonical testers and then executed using ISLA.

Similar (test related) tools to LOTEST have been developed:

CANTEST

The purpose of the tool CANTEST is the derivation of test suites for testing the conformance of LOTOS specifications to a given LOTOS specification using the *conf* relation as given in chapter 2. This tool is able to derive a canonical tester from a *finite LOTOS specification*, along the theory in [Bri 88]. Additional features are:

- derivation of a test suite from an obtained canonical tester, containing deterministic test cases only (i.e., computing the set of irreducible reductions of the canonical tester);
- checking failure equivalence of two specifications;
- checking whether a specification is a (failure) reduction of a given reference specification.

This tool is described in [SEDOS N121] and [Eer 87].

COOPER

This tool has more or less the same functionality as CANTEST, i.e., derivation of a canonical tester based on the *conf* relation, but is based on the compositional way of computing a canonical tester described in [Wez 88] and should therefore be easier to extend to specifications describing infinite behaviours. □

LOTEST offers most of the functionalities found in CANTEST and COOPER in addition to supporting full LOTOS and infinite behaviours. The use of symbolic trees frees the program from the chore of expanding LOTOS behaviours and makes it easier to detect and treat recursion. In the case of basic LOTOS, LOTEST is capable of proving conformance, trace inclusion and failure reduction between execution trees. A detailed description of the algorithms used is given in chapter 4.

The following is a table giving a brief comparison between the three tools:

	CANTEST	COOPER	LOTEST
CO-OP method	No	Yes	Yes
Infinite behaviours	No	No	Yes
Full LOTOS	No	No	Yes
Deriving test cases	Yes	No	No
Proving conformance	No	No	Yes
Failure equivalence	Yes	No	Yes

6.2 Expansion of LOTOS behaviour expressions

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

Milner's *expansion theorem* represents the formalization of this concept. It states that any CCS behaviour expression can be written as a sum (choice) of CCS behaviour expressions, where each expression is written as an action followed by a behaviour expression. In other words, by applying the expansion theorem recursively, it is possible to progressively confine operators different from the sum operator to increasingly more internal sub-expressions. If the given behaviour expression does not contain recursive calls, this will eventually yield to an expression that only contains the sum operator. The expansion theorem applies to LOTOS [ISO 8807], where the sum operator of CCS is the choice operator in the case of LOTOS. The expansion theorem forms the theoretical basis for the generation of symbolic execution trees.

6.3 Using SELA and ISLA to build a symbolic execution tree

SELA and ISLA are used to generate symbolic execution trees from LOTOS specifications. The symbolic expansion process conducted by SELA is divided into three phases [Ash 92]:

Preparation phase

Preparing the proper execution environment involves:

1. Using the LOTOS Translator, to translate the LOTOS source specification into its corresponding internal representation, suitable for use with ISLA.

2. Invoking the ISLA Simulator, to load the translated specification.

Generation phase

The generation phase deals with the generation of the symbolic execution tree from the internal representation of the specification. The tree is created according to the boundaries, namely the depth and the width, specified by the user.

Translation phase

The translation phase corresponds to the generation of the output desired by the specifier for analysis, among one of the following formats:

1. The reduction of the symbolic tree by application of congruence rules.

2. The translation of the symbolic tree into a monolithic style specification.

3. The translation of the symbolic tree into a parametrized tree.

Symbolic trees are saved under two formats: a "prolog" internal form and a text format. LOTEST uses the latter format.

6.4 Some implementation details

The tool LOTEST is implemented in the C programming language. It runs on both the PC (DOS) and the SUN workstation (UNIX) environments. Symbolic trees generated by SELA are parsed by LOTEST and stored in a data structure. LOTEST uses dynamic allocation of memory; so symbolic trees can grow as big as memory permits.

Since symbolic trees are general trees (each node can have any number of children), we used a binary-like structure to represent the general tree. Each node has a pointer to its first *child* and a pointer to its right *sibling*. For performance reasons, we also made each node point to its *father*.

The tool was tested with several examples. In Appendix B, some of these examples are shown. The tool seems to be more stable in the UNIX environment due to its memory requirements; i.e., many of the functions implemented in the tool are recursive and require extensive use of the internal stack which is very limited in size in the DOS environment, where, due to the segmented memory architecture, the stack segment is not allowed to grow bigger than 64K.

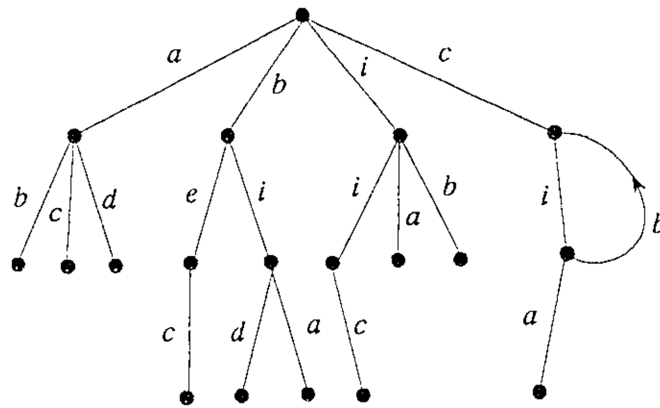
Remark

The word *tree* may seem inappropriate to designate the structure we are using since it may contain cycles. We have chosen the word *tree* for consistency with the previous work. But definitely symbolic trees and failure trees do not have a tree structure.

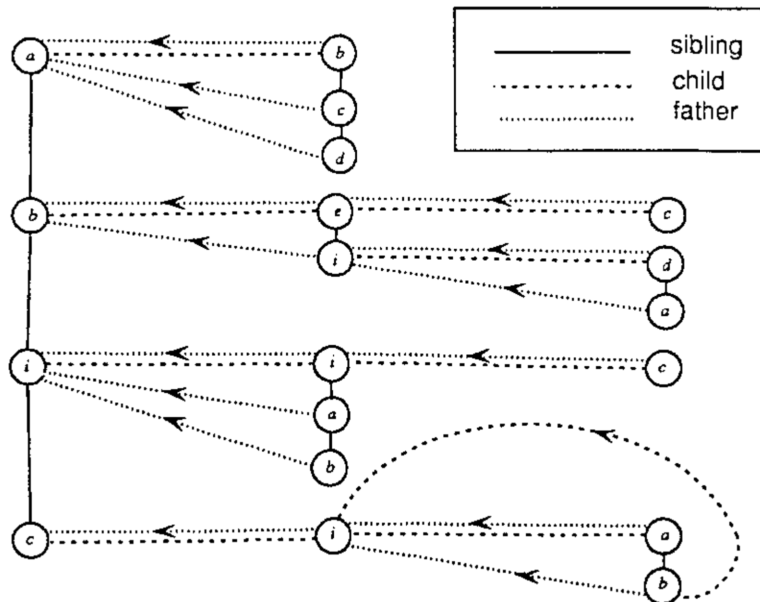
The following example shows a symbolic execution tree and how it is stored in memory by LOTEST.

Example 6.1

Consider the following symbolic execution tree:



It is stored using the following data structure:



The program consists of seven modules:

1. TREE

This module contains the necessary functions for manipulating a symbolic tree. Some of these functions are:

showtree :	displays a tree on the screen.
savetree :	saves a tree in a file.
counttree :	counts how many nodes are there in a tree.
stable :	checks if the root of a tree is *stable*.
findafter :	generates the resulting tree after engaging in a given action.
cleantree :	removes a tree from memory.

2. SETS

This module contains the necessary functions for manipulating sets. Some of these functions are:

showset :	displays a set on the screen.
saveset :	saves a set in a file.
issubset :	checks if a set is a subset of another set.
addset :	adds the contents of set to another set.
buildcoop :	builds the compulsory and the options sets for a given tree.
optimize :	optimizes the compulsory and the options sets.
findout :	finds the Out set of a given tree.
findorth :	finds the orthogonal set of a given set.
findalpha :	finds the alphabet of a given tree.

3. PARSE

This module contains the necessary functions for parsing a file containing a symbolic tree generated by SELA. Some of these functions are:

getsym : is the lexical analyzer. It generates *tokens* to be passed to the parser.

parsetree : is the parser. It parses a text file generated by SELA and builds the data structure containing the symbolic execution tree.

4. TEST

This module contains the necessary functions for building the canonical tester. The most important function in this module is:

buildtest : generates the canonical tester of a symbolic tree.

5. REF

This module contains the necessary functions for generating failure trees, refusal sets and traces.

refusals : finds the refusal sets at the root of a symbolic tree.

trace : finds the traces of a tree up to a given depth.

buildref : builds the failure tree of a symbolic tree.

findgate : finds traces that lead to a given gate name.

6. CONF

This module contains the necessary functions for *proving* conformance between execution trees generated from Basic LOTOS specifications.

7. VIEW

This module contains the necessary functions for LOTEST's user interface (see appendix A for a detailed description of the interface). Some of these functions are:

getcom : the command interpreter of LOTEST.

execom : executes a command.

Chapter 7

Conclusions

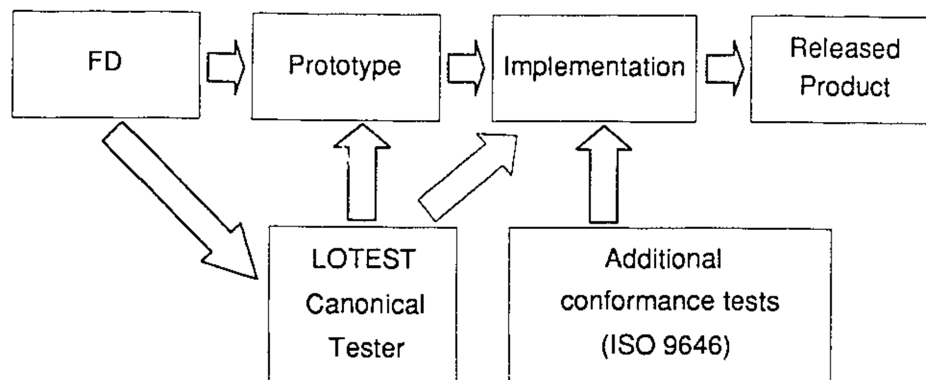
In this thesis, we have summarized the basic theory of conformance testing for LOTOS based applications, we have proposed some generalizations of this theory, and we have described a tool that we have implemented in order to demonstrate the theory and its generalization.

In chapter 2, we have introduced the CO-OP method for Basic LOTOS: a method for the derivation of *canonical testers* [Wez 88]. The way the method has been published in the literature, it can only deal with specifications that do not involve *recursion*. In chapter 3, we have generalized the CO-OP method in such a way as to remove this restriction: the only restriction we have is that the specification must have a *finite expansion* (finite sets of states and transitions). In chapter 4, we proposed an algorithm for *proving conformance* between finite behaviour trees, using the concepts of *complete failure trees* and *canonical trace equivalents*. We finally presented, in chapter 5, the problems encountered while applying the CO-OP method to Full LOTOS behaviour trees and possible solutions for these problems. All the algorithms presented in the thesis were implemented in our tool LOTEST, which will be included in the University of Ottawa LOTOS toolkit.

LOTEST was described in detail in the appendixes and various examples were given. As currently implemented, the results given by the tool can be impractical because behaviour trees and canonical testers can be very large. Future work involves making this tool and the underlying theory more practical, by combining it with methods for compacting the behaviour trees and by implementing heuristics to select useful test cases from the canonical testers. Besides that, we believe that the tool with its diversified functionalities will be very useful for learning LOTOS testing theory and for future applications.

Many of the ideas presented in the thesis couldn't be applied to Full LOTOS due to some weaknesses in the testing theory for Full LOTOS. The concepts of failure trees and refusal sets need to be extended to parameterized labelled transition systems. As we have seen in chapter 5, the solutions proposed for solving some of the problems encountered while trying to extend the testing theory of Basic LOTOS to Full LOTOS are costly and inefficient and therefore leave a lot to be done in future research.

To conclude, it is interesting to reflect about the role that tools and techniques discussed in the thesis, could fulfil in the real world of conformance testing [ISO 9646]. The following diagram presents a possible global view:



Appendix A

A Tool For LOTOS Testing Theory

All the algorithms presented in the previous chapters have been implemented in a tool we call LOTEST. In this appendix we will present the tool, its new features compared to what has been done before, and its limitations.

A.1 A brief overview of LOTEST

LOTEST runs on both the PC and the SUN workstation environments. To activate LOTEST the user must type "LOTEST". A help screen appears showing the list of all available commands. LOTEST is an interactive system; it interacts with the user by means of commands. The LOTEST command line is where the user types commands. The command prompt "? →" indicates that LOTEST is waiting for a command. To direct LOTEST to perform a task, the user types a command and then presses ENTER. An LOTEST command has two parts. Every command has a *command name*. Some commands require one or more *parameters*. The command name states the action the user wants LOTEST to carry out. Some commands consist only of a command name. Parameters define objects the user wants LOTEST to act on.

The following is the first screen shown by LOTEST.

LIST OF COMMANDS:				
Load	fname	[AS tname]	CYcle	tname
Test	tname	[AS tname]	Out	tname
TRace	tname	[AS tname]	View	tname
Save	tname	[AS fname]	REMove	tname
REName	tname	tname	Alpha	tname
COMPare	tname	tname	Refuse	tname
Paths	tname	[depth]	Coop	tname
Find	action	IN tname	CounT	tname
LaBel	label	IN tname As tname	ORth	tname

Build	tname [AS tname] [USing list]
Let	tname = tname (AFter | SKip) trace
SeT	[DEFault | DEPth number | OPTimize | Full]
SeT	[Pause | LiNes number | SYStem]
SYStem	command
ChDir	[path]

LiST [tname]	ReSeT	Help [command]	Quit

? → ■

Screen 1 : Help screen

This screen shows the list of all commands available along with their parameter lists. Note the command prompt at the bottom of the screen.

A.2 Command reference

A.2.1 Syntax conventions

Letters in capital indicate abbreviations of commands; e.g., the command *compare* can also be written as *comp*.

The following is a sample syntax line:

```
SAMple [ arg1 ] [ arg2 | arg3 ] AS arg4
```

The meaning of these elements is as follows:

SAMple	Specifies the name of the command. In this case the command is either *sam* or *sample*
[ ]	Indicates an item that is optional. To include the optional information described within the brackets, type only the information, not the brackets themselves.
|	Separates two or more mutually exclusive choices in a syntax line.
arg1, arg2 ...	Are variable arguments. Variable arguments are in lower case letters.
AS	Is a constant argument.

Spaces delimiting the components of a syntax line can be any sequence of Tabs or Spaces or a combination of both.

LOTEST is not case sensitive except for symbolic tree names or file names.

A.2.2 Online help with commands

LOTEST includes online help for commands. To get help with syntax, parameters, and the purpose of a command, type the following:

```
Help command
```

Typing *Help* with no parameters brings the help screen (see Screen 1).

A.2.3 LOTEST command summary

Commands are presented in the same order as they appear on the help screen (see Screen 1). Note also that some commands can be applied only on symbolic trees generated from Basic LOTOS behaviours.

LOAD

Format : `Load fname [ AS tname ]`

LOTEST does not accept LOTOS specifications directly. These must be transformed into an intermediate model (symbolic execution tree) by the tool SELA before getting loaded by LOTEST. SELA stores symbolic execution trees in disk using two file formats: a prolog internal form and a proprietary text format. LOTEST uses the text format. The *load* command loads a symbolic tree from disk saved under the name *fname* into memory under the name *tname*. If *tname* is not specified, *fname* will be used by default. LOTEST does not impose a limit on the number of symbolic trees loaded at once. All symbolic trees must have unique names.

Example :

`load ex1 as tree1`	loads *ex1* from disk under the name *tree1*
`load ex2`	loads *ex2* from disk under the name *ex2*
`load ex3 as tree1`	causes an error since *tree1* is already loaded

TEST

Format : `Test tname1 [ AS tname2 ]`

Test generates the tester of *tname1* under the name *tname2*. If *tname2* is not specified the tester is named *ttname1* by default.

Example :

`test ex1 as test1`	generates *test1* the tester of *ex1*
`test ex1`	generates the tester of *ex1* under the name *tex1*

TRACE

Basic LOTOS

Format : `TRace tname1 [ AS tname2 ]`

Trace generates a deterministic symbolic tree which is trace equivalent to *tname1*. If specified, the resulting tree is named *tname2*; if not it is named by default *ptname1*.

Example :

`trace ex1 as p1` the resulting tree is named *p1*
`trace ex1` the resulting tree is named *pex1*

SAVE

Format : `Save tname [ AS fname ]`

Save saves the symbolic execution tree named *tname* on disk under the name *fname*. If *fname* is not specified *tname* will be used by default.

RENAME

Format : `REName tname1 tname2`

Rename renames the symbolic tree named *tname1* as *tname2*.

PATHS

Format : `Paths tname [ depth ]`

Paths displays all the possible traces of *tname* up to a given depth. If *depth* is not specified, the default value set by *set depth* command is used.

Example : `paths tree1 10` displays traces of length less or equal to 10

BUILD

Basic LOTOS

Format : `Build tname1 [ AS tname2 ] [ USing list ]`

Build generates the complete failure tree of the symbolic tree named *tname1*. The resulting failure tree is named *tname2* if *tname2* is specified; if not it is named *rtname1*. The alphabet (set of actions) used to generate *tname2* is the union of the alphabets of *tname1* and the alphabets of the symbolic trees listed in *list*.

Example : `build ex1 as tree1 using ex2 ex3 ex4`
The alphabet used to build *tree1* is the union of the alphabets of *ex1*, *ex2*, *ex3* and *ex4*.

FIND

Basic LOTOS

Format : `Find action IN tname`

This command searches for an action in a symbolic tree then displays traces that lead to that action.

Example : `find tea in vending` displays traces that lead to action *tea*

LABEL

Format : `LaBel label IN tname1 AS tname2`

Moves the symbolic tree *tname1* to state *label*. The resulting tree is named *tname2*.

Example : `label bh3 in tree1 as tree2`
Suppose the initial state of *tree1* is *bh0*, the initial state of *tree2* will be *bh3*.

CYCLE

Format : `CYcle tname`

This command checks if *tname* contains a cycle of internal actions.

OUT

Format : `Out tname`

Displays the set *Out(tname)*.

VIEW

Format : `View tname`

Displays on the screen the symbolic execution tree named *tname*.

REMOVE

Format : `REMove tname`

Removes *tname* from memory.

ALPHA

Basic LOTOS

Format : `Alpha tname`

Displays the alphabet of *tname*.

REFUSE

Basic LOTOS

Format : `Refuse tname`

Displays the set *Ref(tname)*.

COOP

Format : `Coop tname`

Displays the sets *Compulsory(tname)* and *Options(tname)*.

COUNT

Format : `CounT tname`

Counts how many actions are there in *tname*. This command can be useful to get an idea of the size of a symbolic execution tree before displaying or printing it.

ORTH

Format : `ORth tname`

Displays the set *Orth(Compulsory(tname))*.

COMPARE

Basic LOTOS

Format : `COMPare tname1 tname2`

LOTEST treats four types of trees: symbolic execution trees, canonical testers of symbolic trees, canonical trace equivalents of symbolic trees and failure trees. Only trace equivalents and failure trees can be used by *compare*. The *compare* command is used to compare two trees of the same type.

In the case of canonical trace equivalents, *compare* checks if the set of traces of *tname1* is included in the set of traces of *tname2*.

Example : In this example, we want to check if two symbolic trees *ex1* and *ex2* are trace equivalents.

```
load ex1
load ex2
trace ex1 as tr1
```

```
trace ex2 as tr2
compare tr1 tr2
compare tr2 tr1
```

is $Tr(tr1) \subseteq Tr(tr2)$?
is $Tr(tr2) \subseteq Tr(tr1)$?

In the case of failure trees, *compare* checks if after any trace accepted by both trees, the sets refused by *tname2* can also be refused by *tname1*.

Example : In the following we want to check if *ex1 conf ex2*, then if *ex1 red ex2*.

```
load ex1
load ex2
build ex1 as fail1 using ex2
build ex2 as fail2 using ex1
trace ex1 as tr1
trace ex2 as tr2
compare fail2 fail1
compare tr1 tr2
```

is *ex1 conf ex2* ?
is $Tr(ex1) \subseteq Tr(ex2)$?

LET

Basic LOTOS

Format :

```
Let  tname1  =  tname2  AFter  trace
Let  tname1  =  tname2  SKip  trace
```

The *let* command finds the behaviour of a symbolic tree after a given trace of actions according to definition 2.7 of chapter 2. It comes under two flavours: the *let after* and the *let skip* commands. The *let after* command is the regular one; it assumes that *tname2* has exactly one known initial state. The *skip after* command assumes that the initial state of *tname2* is unknown by the user.

Example : Suppose *ex1 = a; b; x; stop [] c; a; d; stop [] a; b; stop*

```
let tree1 = ex1 after a
```

⇒ *tree1 = i; b; x; stop [] i; b; stop*

```
let tree2 ex1 skip a
```

⇒ *tree1 = i; b; x; stop [] i; d; stop [] i; b; stop*

```
let tree3 = ex1 after a b
```

⇒ *tree3 = i; x; stop [] i; stop*

```
let tree4 = ex1 after x
```

⇒ *undefined*

SET

Format :

```
SeT  [ DEFault | DEPth number | OPTimize | Full ]
SeT  [ Pause | LiNes number | SYStem ]
```

The *set* command sets up the environment. The following are the meanings of each variable. The default values are specified between parenthesis.

default	Restores the default setup.
depth (5)	Sets the default trace depth for the *paths* command.
optimize (ON)	Sets optimizations to *ON* or *OFF*.
full (ON)	If set to *OFF*, Full LOTOS symbolic trees are treated as Basic LOTOS symbolic trees (only gate names are treated; experiments are ignored).
pause (ON)	If set to *ON*, the display pauses after the number of lines specified by the *lines* default value.
lines (20)	Sets the default value for *lines*.
system (ON)	If set to *ON*, each command not accepted by LOTEST is returned to the operating system.

Example :

```
set default
set depth 20
set optimize off
set full on
set lines 22
set system off
```

SYSTEM

Format :

```
SYStem  command
```

Passes *command* to the operating system's command interpreter.

Example :

`system vi ex1`	calls the *vi* editor *(UNIX)*
`system dir /p`	lists the contents of the current directory *(DOS)*

CHDIR

Format : `ChDir [ path ]`

Changes the current directory if *path* is specified. If not it displays the current working directory.

LIST

Format : `LiST [ tname ]`

If *tname* is not specified this command lists all trees currently present in memory. If *tname* is specified, *list* gives information about *tname*. For trees derived using the *let* command, the initial tree is displayed.

RESET

Format : `ReSeT`

Clears the memory. All information loaded in memory is lost. Default settings are not reset by this command.

HELP

Format : `Help [ command ]`

If *command* is specified, help about *command* is displayed. If *command* is not specified this command displays the help screen (see Screen 1).

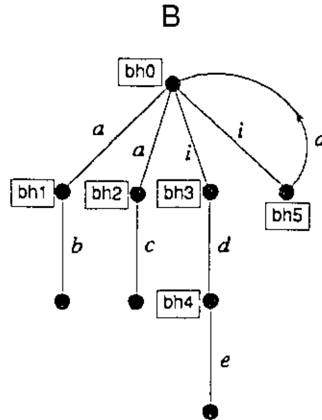
QUIT

Format : `Quit`

Exits from LOTEST.

A.2.4 Format of the trees used by LOTEST

Consider the following symbolic execution tree:



For this tree to be used by LOTEST, it must be saved in a text file under the following format. That's also the format of the files generated by SELA.

```
bh0 * 1 a
bh1 * | 1 b
    * 2 a
bh2 * | 1 c
    * 3 i
bh3 * | 1 d
bh4 * | | 1 e
    * 4 i
bh5 * | 1 d ==> again bh0
```

That format is also used to display and save to disk symbolic execution trees and canonical testers.

For failure trees, each state (or label) in the tree is preceded by its corresponding refusal set. The failure tree of B is:

```
{{a,b,c,e}}
bh0 * 1 a
{{a,c,d,e},{a,b,d,e}}
bh1 * | 1 b
    * | 2 c
    * 2 d
{{a,b,c,d},{a,b,c,e}}
bh2 * | 1 e
    * | 2 a ==> again bh1
    * | 3 d ==> again bh2
```

A.3 Limitations

As we have seen, using the symbolic execution tree as an intermediate model has many advantages. In addition to making the task of extracting test cases easier, it allowed us to treat many cases of LOTOS behaviours that were not treated before. LOTEST is capable of treating recursive LOTOS behaviours and a big subset of Full LOTOS behaviours. Most of the limitations imposed by LOTEST are those imposed by the symbolic execution tree model.

A.3.1 Size of the symbolic execution tree

Replacing the *interleave* operator of LOTOS by the *choice* operator causes the symbolic tree to explode in size. Large symbolic trees are memory "hungry" and they cause slow performance problems. SELA fixes the problem by cutting the tree in both width and depth. This approach causes LOTEST to be unable to treat some cases of LOTOS behaviours that make extensive use of the parallel composition operators.

A.3.2 Non tail recursion

In the process of building the symbolic execution tree, SELA replaces recursion by cycles (or loops). This is not always feasible. SELA is able to treat only the case of tail

recursion. If not the tree will be cut.

Example A.1

Consider the following recursive behaviour expression:

B = (a; B >> b; exit) [] exit

B can never be represented by a finite state LTS, as traces of B have the following format:

$a^n b^n$ where $n \in \{0, 1, ...\}$

A.3.3 Full LOTOS

LOTEST does not fully support Full LOTOS as discussed in chapter 5.

Appendix B

Examples

B.1 A working session with LOTEST

This is an actual output from LOTEST:

```
=====================================================================
                       LIST OF COMMANDS:
=====================================================================
 Load     fname   [ AS tname ]           |         CYcle     tname
 Test     tname   [ AS tname ]           |         Out       tname
 TRace    tname   [ AS tname ]           |         View      tname
 Save     tname   [ AS fname ]           |         REMove    tname
 REName   tname   tname                  |         Alpha     tname
 COMPare  tname   tname                  |         Refuse    tname
 Paths    tname   [ depth ]              |         Coop      tname
 Find     action  IN tname               |         CounT     tname
 LaBel    label   IN tname AS tname      |         ORth      tname
---------------------------------------------------------------------
 Build    tname [ AS tname ] [ USing list ]
 Let      tname  =  tname  ( AFter | SKip )  trace
 SeT      [ DEFault | DEPth number | OPTimize | Full ]
 SeT      [ Pause   | LiNes number | SYStem   ]
---------------------------------------------------------------------
 SYStem command                          |         ChDir  [ path ]
---------------------------------------------------------------------
 LiST [ tname ]        ReSeT        Help [ command ]          Quit
=====================================================================

? -> load ex33 as tree1

***** parsing ...

bh0   * 1 a
bh1   * | 1 b
bh3   * | | 1 a ==> again bh1
      * | 2 i
bh4   * | | 1 b
      * | | 2 a
```

```
      * 2 a
bh5   * | 1 b
bh6   * | | 1 a
      * | | 2 b ==> again bh0

***** end of parsing

? -> set

Current settings:

Maximum trace depth : 5
Optimizations       : ON
Full LOTOS          : ON
System              : ON
Pause               : ON
Pause after         : 20 line(s)

? -> set depth 10

done

? -> paths tree1

1> a b a b a b a b a b
2> a b a b a b a b a a
3> a b a b a b a a
4> a b a b a a
5> a b a a
6> a b b a b a b a b a
7> a b b a b a b a a
8> a b b a b a a
9> a b b a b b a b a b
10> a b b a b b a b a a
11> a b b a b b a b b a
12> a b b a b b a a
13> a b b a a
14> a a

? -> refuse tree1

The Refusal set is:
{{b}}

The Reduced Acceptance set is:
{{a}}

? -> build tree1 as fail1

done

? -> view fail1

{{b}}
bh0   * 1 a
{{a}}
bh1   * | 1 b
{{a,b}}
bh2   * | | 1 a
{{a,b}}
bh3   * | | | 1 b
{{a,b}}
bh4   * | | | | 1 a
{{}}
bh5   * | | | | | 1 b ==> again bh4
```

```
      * | | | | | 2 a
      * | | | 2 a
      * | | 2 b ==> again bh0
      * | 2 a

? -> test tree1 as test1

done

? -> view test1

bh0   * 1 a
bh1   * | 1 i
bh2   * | | 1 b
bh3   * | | | 1 i
      * | | | 2 a
bh4   * | | | | 1 i
      * | | | | 2 b
bh5   * | | | | | 1 i
      * | | | | | 2 a
bh6   * | | | | | | 1 i
bh7   * | | | | | | | 1 b ==> again bh5
      * | | | | | | 2 i
bh8   * | | | | | | | 1 a
      * | | | | | | 3 b ==> again bh5
      * | | | | 3 a
      * | | | 3 b ==> again bh0
      * | 2 b ==> again bh3
      * | 3 a

? -> test test1 as tree2

done

? -> list

=============================================================
 Tree name          |          Type          |        Initial
=============================================================
   tree1            |   original  tree    |
   fail1            |    refusal tree     |
   test1            |  canonical tester   |
   tree2            |  canonical tester   |
=============================================================

? -> quit

Are you sure ? y
```

B.2 Checking interesting properties with LOTEST

B.2.1 The tester of the tester

Using LOTEST we can check whether the following property is true on a given example:

$$T(T(B)) \approx B$$

```
? -> load ex45 as a

***** parsing ...

bh0   * 1 i
bh1   * | 1 ConReq
bh2   * | | 1 ConCnf
bh3   * | | | 1 i
bh4   * | | | | 1 i
bh5   * | | | | | 1 DatReq ==> again bh4
      * | | | | | 2 i
bh6   * | | | | | | 1 DisReq
bh7   * | | | | | | | 1 i ==> again bh0
      * | | | | | 3 DisInd ==> again bh7
      * | | | | 2 DatInd ==> again bh4
      * | | | | 3 i ==> again bh6
      * | | | | 4 DisInd ==> again bh7
      * | | 2 DisInd
bh8   * | | | 1 i ==> again bh1
      * | | | 2 ConInd
bh9   * | | | | 1 i
bh10  * | | | | | 1 ConRes ==> again bh3
      * | | | | 2 i
bh11  * | | | | | 1 DisReq ==> again bh8
      * 2 ConInd ==> again bh9

***** end of parsing

? -> test a

done

? -> list

==========================================================
 Tree name         |        Type         |         Initial
==========================================================
   a               |   original  tree    |
   ta              |  canonical tester   |
==========================================================

? -> test ta

done

? -> list

==========================================================
 Tree name         |        Type         |         Initial
==========================================================
   a               |   original  tree    |
   ta              |  canonical tester   |
   tta             |  canonical tester   |
==========================================================

? -> build a

done

? -> build tta
```

```
done

? -> list

=========================================================================
 Tree name          |         Type          |              Initial
=========================================================================
   a                |   original  tree      |
   ta               |  canonical tester     |
   tta              |  canonical tester     |
   ra               |    refusal tree       |
   rtta             |    refusal tree       |
=========================================================================

? -> compare ra rtta

rtta conforms to ra

? -> compare rtta ra

ra conforms to rtta

? -> trace a

done

? -> trace tta

done

? -> list

=========================================================================
 Tree name          |         Type          |              Initial
=========================================================================
   a                |   original  tree      |
   ta               |  canonical tester     |
   tta              |  canonical tester     |
   ra               |    refusal tree       |
   rtta             |    refusal tree       |
   pa               |  trace equivalent     |
   ptta             |  trace equivalent     |
=========================================================================

? -> compare pa ptta

ptta is trace included in pa

? -> compare ptta pa

pa is trace included in ptta
```

The symbolic execution tree used in this example was generated from the following Basic LOTOS specification of a simplified transport service handler given by [BB 87].

```
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
```

B.2.2 Refusal sets

LOTEST can be used to check whether a given process is non-deterministic by using the following property:

> A process is non-deterministic if and only if at least one maximal refusal set in its complete failure tree contains more than one set.

The symbolic execution tree used in this example was taken from the same Basic LOTOS specification given in the previous example.

```
? -> load ex45 as tree

***** parsing ...

bh0   * 1 i
bh1   * | 1 ConReq
bh2   * | | 1 ConCnf
bh3   * | | | 1 i
bh4   * | | | | 1 i
bh5   * | | | | | 1 DatReq ==> again bh4
      * | | | | | 2 i
bh6   * | | | | | | 1 DisReq
bh7   * | | | | | | | 1 i ==> again bh0
      * | | | | | | 3 DisInd ==> again bh7
      * | | | | 2 DatInd ==> again bh4
      * | | | | 3 i ==> again bh6
      * | | | | 4 DisInd ==> again bh7
      * | | 2 DisInd
bh8   * | | | 1 i ==> again bh1
      * | | | 2 ConInd
bh9   * | | | | 1 i
bh10  * | | | | | 1 ConRes ==> again bh3
      * | | | | 2 i
bh11  * | | | | | 1 DisReq ==> again bh8
      * 2 ConInd ==> again bh9

***** end of parsing

? -> build tree as fail

done

? -> view fail

{{ConCnf,DatReq,DisReq,DisInd,DatInd,ConInd,ConRes}}
bh0   * 1 ConReq
{{ConReq,DatReq,DisReq,DatInd,ConInd,ConRes}}
bh1   * | 1 ConCnf
{{ConReq,ConCnf,DatReq,DisInd,DatInd,ConInd,ConRes}}
bh2   * | | 1 DatReq ==> again bh2
```

```
      * | | 2 DisReq
{{ConCnf,DatReq,DisReq,DisInd,DatInd,ConInd,ConRes}}
bh3   * | | | 1 ConReq
{{ConReq,DatReq,DisReq,DatInd,ConInd,ConRes}}
bh4   * | | | | 1 ConCnf
{{ConReq,ConCnf,DatReq,DisInd,DatInd,ConInd,ConRes}}
bh5   * | | | | | 1 DatReq
{{ConReq,ConCnf,DatReq,DisInd,DatInd,ConInd,ConRes}}
bh6   * | | | | | | 1 DatReq ==> again bh6
      * | | | | | | 2 DisReq ==> again bh3
      * | | | | | | 3 DisInd ==> again bh3
      * | | | | | | 4 DatInd ==> again bh6
      * | | | | | 2 DisReq ==> again bh3
      * | | | | | 3 DisInd ==> again bh3
      * | | | | | 4 DatInd ==> again bh6
      * | | | | 2 DisInd
{{ConCnf,DatReq,DisReq,DisInd,DatInd,ConInd,ConRes}}
bh7   * | | | | | 1 ConReq ==> again bh4
      * | | | | | 2 ConInd
{{ConReq,ConCnf,DatReq,DisReq,DisInd,DatInd,ConInd},
{ConReq,ConCnf,DatReq,DisInd,DatInd,ConInd,ConRes}}
bh8   * | | | | | | 1 ConRes ==> again bh5
      * | | | | | | 2 DisReq ==> again bh7
      * | | | 2 ConInd ==> again bh8
      * | | 3 DisInd ==> again bh3
      * | | 4 DatInd ==> again bh2
      * | 2 DisInd ==> again bh0
      * 2 ConInd
{{ConReq,ConCnf,DatReq,DisReq,DisInd,DatInd,ConInd},
{ConReq,ConCnf,DatReq,DisInd,DatInd,ConInd,ConRes}}
bh9   * | 1 ConRes ==> again bh2
      * | 2 DisReq ==> again bh0
```

We can see that the refusal sets associated with states: *bh8* and *bh9* of the complete failure tree contain more than one set. We conclude that the specification of the simplified transport service handler is non-deterministic.

B.3 A Full LOTOS example

In this example, the symbolic execution tree of a Full LOTOS specification of a simplified data link service provider was generated using SELA. Then LOTEST was used to derive the general canonical tester of the specification.

Consider the LOTOS specification [QFP 88]. It describes a data link service provider, which uses the alternating bit protocol.

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

library  Boolean endlib

type sequeceNumber is Boolean
       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
        sorts bitString
        opns  empty  :                              ->  bitString
              equal  : bitString,bitString          ->  Bool

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

type Frame is Boolean
        sorts Frame
        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

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
```

The symbolic execution tree of the specification is the following:

```
bh0  * 1 get ?bitString@1:bitString
bh1  * | 1 send !info !0 !bitString@1
bh2  * | | 1 i (specified explicitly)
bh3  * | | | 1 i (hiding: tout)
bh4  * | | | | 1 send !info !0 !bitString@1  ==> again bh2
     * | | 2 receive !ack !(inc(0)) !empty
             ([ack=info] and [inc(0)=0] and [empty=bitString@1])
bh5  * | | | 1 i (enable: exit)
bh6  * | | | | 1 get ?bitString@2:bitString  ==> again bh1
     * | | 3 receive !info !0 !bitString@1
bh7  * | | | 1 [0=0] give !bitString@1
bh8  * | | | | 1 send !ack !(inc(0)) !empty
bh9  * | | | | | 1 i (specified explicitly)
bh10*  | | | | | | 1 i (hiding: tout)
bh11*  | | | | | | | 1 send !info !0 !bitString@1 ==> again bh9
     * | | | | | 2 receive !ack !(inc(0)) !empty
bh12*  | | | | | | 1 i (enable: exit)
bh13*  | | | | | | | 1 get ?bitString@3:bitString
bh14*  | | | | | | | | 1 send !info !(inc(0)) !bitString@3  ==> again bh9
     * | | | | | 3 receive !info !(inc(0)) !empty [info=ack]   ==> again bh7
     * | | | 2 [inc(0)=0] send !ack !(inc(0)) !empty
bh15*  | | | | 1 i (specified explicitly)
bh16*  | | | | | 1 i (hiding: tout)
bh17*  | | | | | | 1 send !info !0 !bitString@1 ==> again bh15
     * | | | | 2 receive !ack !(inc(0)) !empty
```

```
bh18* | | | | | 1 i {enable: exit}
bh19* | | | | | | 1 get ?bitString@3:bitString
bh20* | | | | | | | 1 send !info !(inc(0)) !bitString@3  ==> again bh15
    * | | | | 3 receive !info !(inc(0)) !empty [info=ack]  ==> again bh7
```

The following is the general canonical tester generated by LOTEST. Note that the symbol "#" is used instead of "$\dagger$" because the latter is not an ASCII character.

```
bh0   * 1 get #bitString@1:bitString
bh1   * | 1 send !info !0 !bitString@1
bh2   * | | 1 i
bh3   * | | | 1 send !info !0 !bitString@1 ==> again bh2
      * | | 2 receive !ack !(inc(0)) !empty
                ([ack=info] and [inc(0)=0] and [empty=bitString@1])
bh4   * | | | 1 get #bitString@2:bitString ==> again bh1
      * | | 3 receive !info !0 !bitString@1
bh5   * | | | 1 i
bh6   * | | | | 1 give !bitString@1 [0=0]
bh7   * | | | | | 1 send !ack !(inc(0)) !empty
bh8   * | | | | | | 1 i
bh9   * | | | | | | | 1 send !info !0 !bitString@1 ==> again bh8
      * | | | | | | 2 receive !ack !(inc(0)) !empty
bh10  * | | | | | | | 1 get #bitString@3:bitString
bh11  * | | | | | | | | 1 send !info !(inc(0)) !bitString@3 ==> again bh8
      * | | | | | | 3 receive !info !(inc(0)) !empty [info=ack] ==> again bh5
      * | | | 2 i
bh12  * | | | | 1 send !ack !(inc(0)) !empty [inc(0)=0]
bh13  * | | | | | 1 i
bh14  * | | | | | | 1 send !info !0 !bitString@1 ==> again bh13
      * | | | | | 2 receive !ack !(inc(0)) !empty
bh15  * | | | | | | 1 get #bitString@3:bitString
bh16  * | | | | | | | 1 send !info !(inc(0)) !bitString@3 ==> again bh13
      * | | | | | 3 receive !info !(inc(0)) !empty [info=ack] ==> again bh5
```

Note that our tool, in conjunction with SELA, was able to treat this not completely trivial Full LOTOS example. Note also that the corresponding symbolic execution tree does not show any of the problems discussed in chapter 5; i.e., it does not contain neither overlapping transitions nor internal events with predicates, so the tester produced by the tool is a general canonical tester.

Bibliography

[Ash 92] P. Ashkar. Symbolic Execution of LOTOS Specifications. M.sc. Thesis, University of Ottawa, (1992).

[BALT 90] E. Brinksma, R. Alderden, R. Langerak, J.v.d Lagemaat, J. Tretmans. A formal approach to Conformance Testing, in: J. de Meer (ed.), Proc, 2nd International Workshop on *Protocol Test Systems*, NL, (1990).

[BB 87] B. Bolognesi, E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN systems*, (1987) 14:25-59.

[BeKl 85] J. A. Bergstra, J. W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37 (1), (1985), 77-121.

[Bri 87] E. Brinksma. On the Existence of Canonical Testers. Memorandum INF-87-5, University of Twente, NL, (1987).

[Bri 88] E. Brinksma. A Theory for the Derivation of Tests. In S. Aggerwal, K. Sabani (eds.), *Protocol Specification, Testing and Verification VIII* (North-Holland, 1988). Also: Memorandum INF-88-19, University of Twente, NL, (1989).

[BSS 87] E. Brinksma, G. Scollo, C. Steenbergan. LOTOS Specifications, their implementation and their tests. In S. Sarikaya and G.V Bockmann (eds.), *Protocol Specification, Testing and Verification VI* (North-Holland, 1987), 349-360.

[Doo 91] P. Doornbosch. Test derivation for full LOTOS. M.sc. Thesis, University of Twente, NL, (April 1991).

[Eer 87] H. Eertink. The Implementation of a Test Derivation Algorithm. Memorandum INF-87-36, University of Twente, Enschede, NL, (1987).

[EM 85] B. Ehrig, B. Mahr. Fundamentals of Algebraic Specifications. Springer-Verlag, (1985).

[FP 88] D. Freestone, D. Pitt. A Formal Approach to Conformance Testing. Uk, submitted for publication, (1988).

[Gal 89] S. Gallouzi. Trace analysis of LOTOS behaviours. M. sc. Thesis, University of Ottawa, (1989).

[GHL 88] R. Guillemot, M. Haj-Hussein, L. Logrippo. Executing Large LOTOS Specifications. In S. Aggarwal and K. Sabnani, (eds.), *Protocol Specification, Testing and Verification VIII*, (North-Holland, 1988), 399-410.

[Ghr 92] B. Ghribi. A model checker for LOTOS. M.sc. Thesis, University of Ottawa, (1992).

[GL 89] R. Guillemot, 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.

[GLO 91] S. Gallouzi, L. Logrippo, A. Obaid. A Hoare-style proof system for LOTOS. In *Formal Description Techniques III*, North-Holland (1991).

[HH 89] M. Haj-Hussein. ISLA: An Interactive System for LOTOS Applications. M.sc. Thesis, University of Ottawa, (1989).

[Hoa 85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, (1985).

[ISO 7498] ISO, IS 7498. *Open Systems Interconnection* - Basic Reference Model, (1984).

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

[ISO 9646] ISO/IEC IS 9646. *Open Systems Interconnection* - Conformance testing methodology and framework, (March 1991).

[Led 90] G. Leduc. On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS. Aggregation thesis, Université de Liège, (July 1990).

[LFH 92] L. Logrippo, M. Faci, M. Haj-Hussein. An introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN systems*, 23 (5) (1992) 325-342.

[Mil 80] R. Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, (Springer-Verlag, 1980) No. 92, Springer-Verlag, (1980).

[Mil 89] R. Milner. Communication and Concurrency. Prentice Hall (1989).

[Mye 79] G. L. Myers. The Art of Software Testing (Wiley, 1979).

[Pit 87] D.H. Pitt. A Formal Approach to Conformance Testing. FORMA WP B69, GB, (1987).

[PM 90] L. Probert, O. Monkewich. TTCN - The International Notation For Conformance Testing of Communication Systems.

[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.

[SEDOS N121] ESPRIT/SEDOS/N121, Towards Practical Verification of LOTOS Specifications, (October 1987).

[Ste 86] C. Steenbergan. Conformance Testing of OSI Systems. Memorandum INF-86-24. University of Twente, NL (August 1986).

[TrVe 92] J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communications. In *Protocol, Testing, and Verification XII*, (1992).

[vE 89] P. van Eijk. The Design of Simulator Tool. in P. van Eijk et al., (eds.), *The Formal Description Technique LOTOS*, (North-Holland, Amsterdam, 1989) 351-390.

[Wez 88] C. D. Wezeman. The CO-OP method, A method for Compositional Derivation of Conformance Testers. M.sc. Thesis, University of Twente, NL, (August 1986).

[Wez 89] C. D. Wezeman. The CO-OP method, A method for Compositional Derivation of Canonical Testers. In Proc. IFIP WG6.1 Ninth International Symposium on *Protocol Specification, Testing and Verification*, Enschede, Netherlands, (June 1989) 145-158.

[Wez 90] C. D. Wezeman. Data typing the CO-OP method. In: Deliverable 1990 ESPRIT Project 2304 LOTOSPHERE, Task 1.3 *Testing* Lo/WP1/T1.3/N0006/V5.

[WBL 91] C. D. Wezeman, S. Batley, and J. A. Lynch. Formal methods to assist Conformance Methods, a case study. In *Formal Description Techniques III*, North-Holland (1991).