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

Thesis Committee

Supervisor:

Dr. Luigi Logrippo, University of Ottawa.

Examiners:

Dr. Abdellatif Obaid, Université de Quebec à Hull.

Dr. Robert L. Probert, 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

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 : $\rightarrow$ Nat
    succ : Nat $\rightarrow$ 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 : $\rightarrow$ Nat
    succ : Nat $\rightarrow$ 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 [g_1, g_2, ... g_n] (v_1, v_2, ... v_m): functionality
behaviour
< behaviour expression >
where
< process definitions >
endspec

A process is viewed as a black box interacting with its environment via its observable gates. Its internal actions are unobservable by the environment. The behaviour expression is built by 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*.

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

$E_1, \ldots E_m$ 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.

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

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.

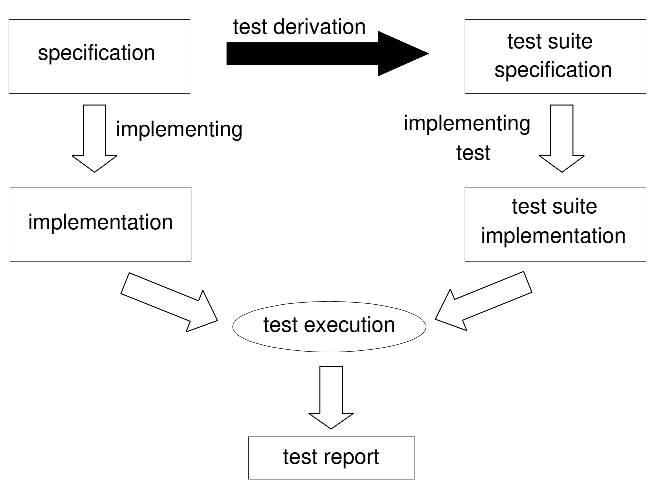


Figure 1.1 : The importance of test derivation

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.

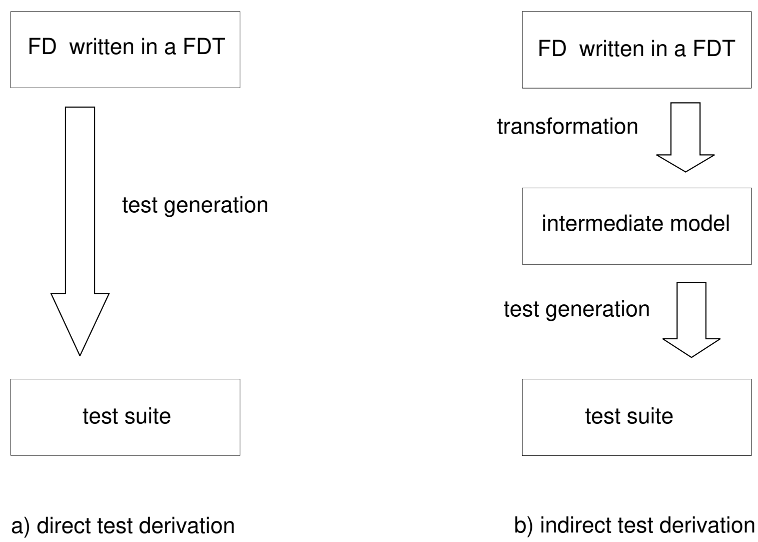


Figure 1.2 : Direct and indirect test derivation

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 = { $<s_1, a, s_2>$ | $s_1, s_2 \in$ Stat and $a \in$ Act } is a set of transitions.
4. s_0 is the initial state. □

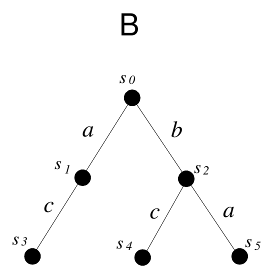
A transition in a LTS is a 3-tuple < s_1, a, s_2 > $\in$ Trans, usually represented by: $s_1 -a\rightarrow s_2$, and we have by definition: $< s_1, s_2 > \in -a\rightarrow$.

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

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:

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

$\text{Act} = \{a, b, c, i\}$

$L = \{a, b, c\}$

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

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

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

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

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
$\eta, \eta_1, \dots \eta_n$	are actions from the set Act
$\mu, \mu_1, \dots \mu_n$	are actions from the set L
$B, C, B_1, \dots B_n$	are labelled transition systems
σ, σ'	are strings of actions from the set L
$B-\eta\rightarrow C$	$B = < Stat, Act, Trans, s_B >$; $C = < Stat, Act, Trans, s_C >$ and $< s_B, \eta, s_C > \in$ Trans
$B-\eta\rightarrow$	$\exists C : B-\eta\rightarrow C$
$B\not-\eta\rightarrow$	$\nexists C : B-\eta\rightarrow C$
$B-\eta_1 \dots \eta_n\rightarrow C$	$\exists B_i \ (1 \le i \le n) : B = B_0-\eta_1\rightarrow B_1-\eta_2\rightarrow \dots -\eta_n\rightarrow B_n = C$
$B-i^0\rightarrow C$	$B = C$
$B=\varepsilon\Rightarrow C$	$\exists n \ge 0 : B-i^n\rightarrow C$
$B=\mu\Rightarrow C$	$\exists B_1, B_2 : B=\varepsilon\Rightarrow B_1-\mu\rightarrow B_2=\varepsilon\Rightarrow C$
$B=\mu_1 \dots \mu_n\Rightarrow C$	$\exists B_i \ (1 \le i \le n) : B = B_0=\mu_1\Rightarrow B_1=\mu_2\Rightarrow \dots =\mu_n\Rightarrow B_n = C$
$B=\mu_1 \dots \mu_n\Rightarrow$	$\exists C : B=\mu_1 \dots \mu_n\Rightarrow C$
$B\not=\sigma\Rightarrow$	$\nexists C : B=\sigma\Rightarrow C$
$B=\sigma\rightarrow C$	if $\sigma = \varepsilon$ then $B = C$ if $\sigma = \sigma'\mu$ then $\exists B' : B=\sigma'\Rightarrow B'-\mu\rightarrow C$
$\Sigma_{j \in \{1, \dots n\}} B_j$	B_1 [] B_2 [] ... 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)	$B\not-i\rightarrow$
B refuses (σ, A)	$\exists B' : (B=\sigma\Rightarrow B'$ and $\forall a \in A : B'\not=a\Rightarrow)$

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 conf S $\Leftrightarrow$ $\forall\sigma \in$ Tr(S), $\forall A \subseteq L$: if I refuses(σ, A) then S refuses(σ, 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 $\prod_{conf}(S)$ consisting of one element called the *canonical tester*. A formal definition of a canonical tester will be given later in the chapter.

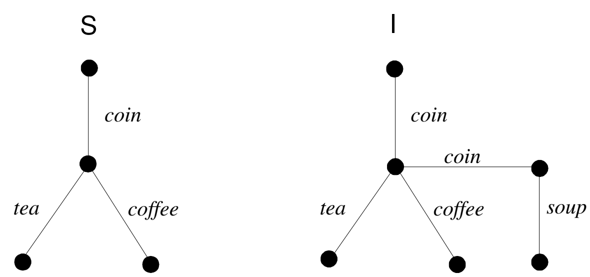
The *conf* relation is not transitive; so if S_3 *conf* S_2 and S_2 *conf* S_1, we do not have necessarily S_3 *conf* S_1.

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)

$$\text{I red S} \iff \forall \sigma \in L^*, \forall A \subseteq L : \quad \text{if I refuses}(\sigma, A) \text{ then S refuses}(\sigma, A).$$ □

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

Definition 2.4 (testing equivalence)

$$B_1 \approx B_2 \iff B_1 \text{ red } B_2 \text{ and } B_2 \text{ red } B_1.$$ □

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.

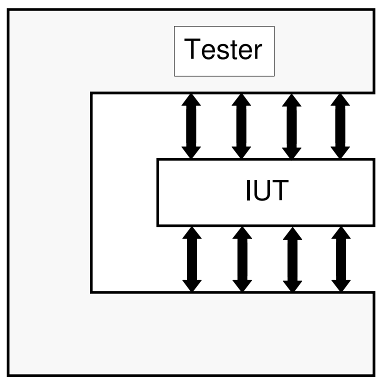


Figure 2.1 : The Utilized Testing Architecture

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)

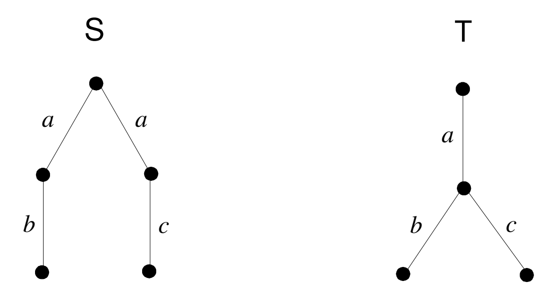
Let S be a process. A (general) canonical tester of the process specification S is a process T(S) that satisfies:

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

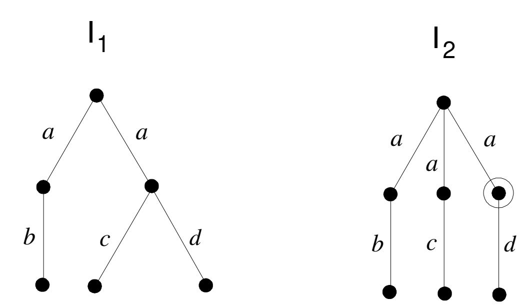
(ii) For all processes P, P conf S iff
for all $\sigma \in L^*$, $P \| T(S) = \sigma \Rightarrow B$ and $B \approx$ stop implies $T(S) = \sigma \Rightarrow B'$ and $B' \approx$ stop

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

$$\begin{array}{lll} T(B) & = & \sum_{V \in \text{Orth(Compulsory(B))}} i; \Sigma_{a \in V} a; \ldots \\ & [] & \sum_{a \in \text{Options(B)}} a; \ldots \end{array}$$

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

$$\begin{array}{lll} T(B) & = & i; stop \\ & [] & \sum_{a \in \text{Out(B)}} a; \ldots \end{array}$$

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 Ø ∉ Compulsory(B) then

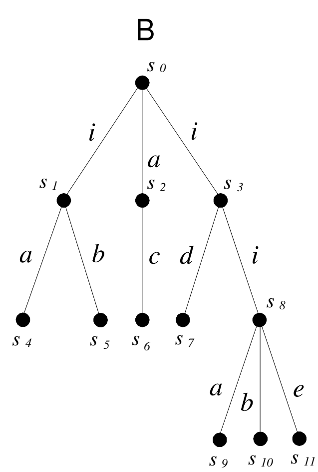
$$T(B) = \sum_{V \in Orth(Compulsory(B))} i; \sum_{a \in V} a; T(B \text{ after } a)$$
$$[] \sum_{a \in Options(B)} a; T(B \text{ after } a)$$

else { Ø ∈ Compulsory(B) }

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

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)

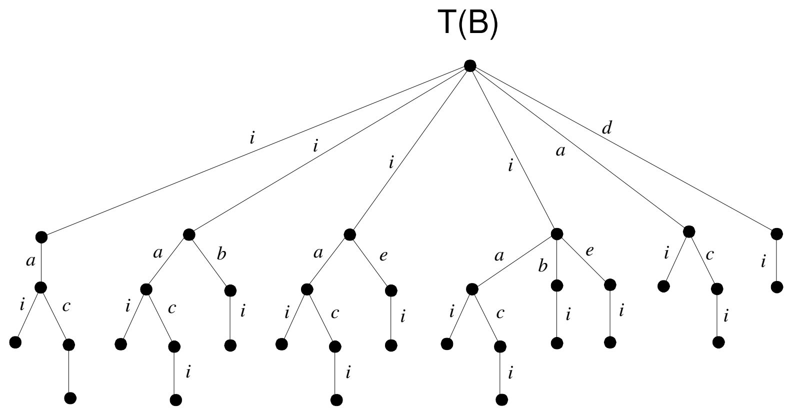
Compulsory(B_2) = {Ø}

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}}

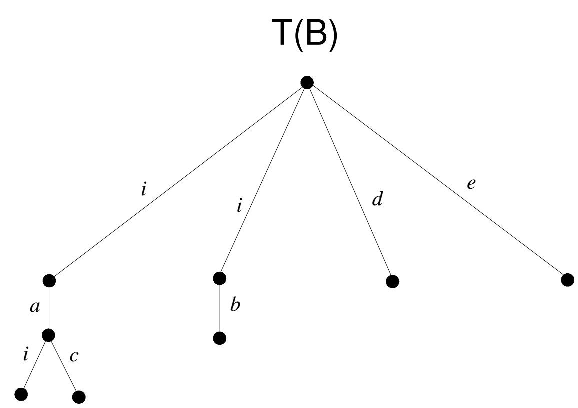
Options(B) = {d, e} Rules 1 and 3

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:

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

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)

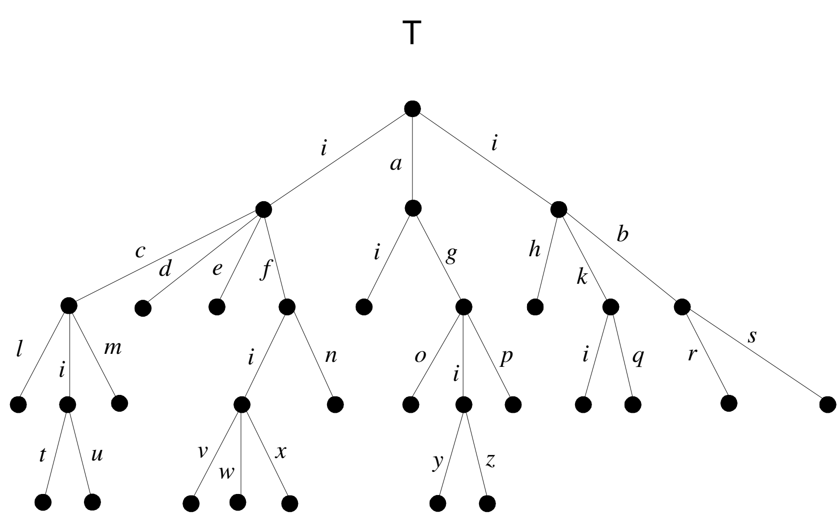
T_4 = a; (i; stop [] c; stop) [] d; stop [] e; stop

T_5 = b; stop [] d; stop [] e; stop □

The following is a more complex example:

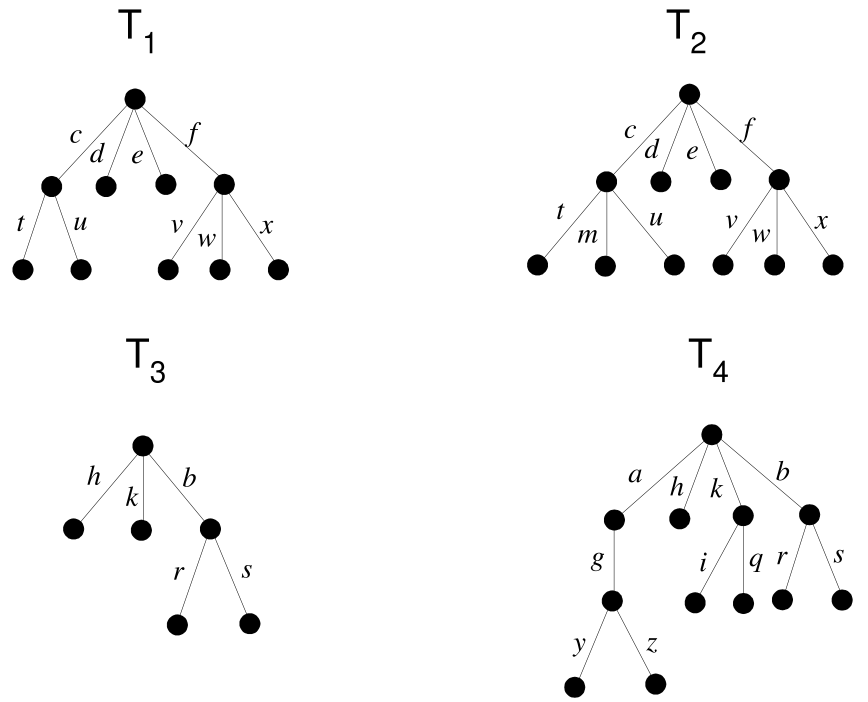
Example 2.7

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

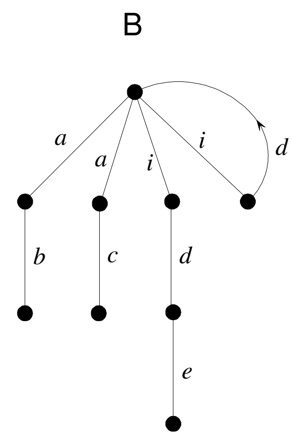
3.1.1 Recursive behaviour expressions

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: $B\ \mathit{after}\ \mu$ where $B\ \mathit{after}\ \mu = \sum_{B'\ :\ B=\mu\rightarrow B'} i;\ B'$. 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) = Ø

Compulsory(B_2) = {{d}, {e}}

Orth(Compulsory((B_2)) = {d, e}

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

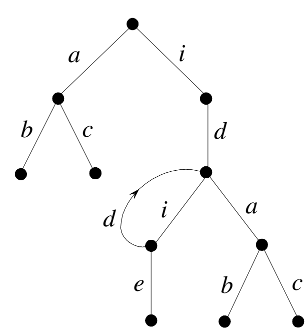
Since we can easily see that B_3 = B_2, we can stop the recursive process and write:

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
$s{-}\eta{\to}s'$	$< s, \eta, s' >$ is a transition
$s{-}\eta{\to}$	$\exists s' : s{-}\eta{\to}s'$
$s{\not-}\eta{\to}$	$\nexists s' : s{-}\eta{\to}s'$
$s{-}\eta_1\ldots\eta_n{\to}s'$	$\exists s_i\ (\ 1 \le i \le n\ ) : s = s_0{-}\eta_1{\to}s_1{-}\eta_2{\to}\ldots{-}\eta_n{\to}s_n = s'$
$s{-}i^0{\to}s'$	$s = s'$
$s{=}\varepsilon{\Rightarrow}s'$	$\exists n \ge 0 : s{-}i^n{\to}s'$
$s{=}\mu{\Rightarrow}s'$	$\exists s_1, s_2 : s{=}\varepsilon{\Rightarrow}s_1{-}\mu{\to}s_2{=}\varepsilon{\Rightarrow}s'$
$s{=}\mu_1\ldots\mu_n{\Rightarrow}s'$	$\exists s_i\ (\ 1 \le i \le n\ ) : s = s_0{=}\mu_1{\Rightarrow}s_1{=}\mu_2{\Rightarrow}\ldots{=}\mu_n{\Rightarrow}s_n = s'$
$s{=}\mu_1\ldots\mu_n{\Rightarrow}$	$\exists s' : s{=}\mu_1\ldots\mu_n{\Rightarrow}s'$
$s{\neq}\sigma{\Rightarrow}$	$\nexists s' : s{=}\sigma{\Rightarrow}s'$
$s{=}\sigma{\to}s'$	if $\sigma = \varepsilon$ then $s = s'$ if $\sigma = \sigma'\mu$ then $\exists s'' : s{=}\sigma'{\Rightarrow}s''{-}\mu{\to}s'$
Out(s)	$\{\ a \in L \mid s{=}a{\Rightarrow}\ \}$
Tr(s)	$\{\ \sigma \in L^* \mid s{=}\sigma{\Rightarrow}\ \}$
stable(s)	$s{\not-}i{\to}$
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.

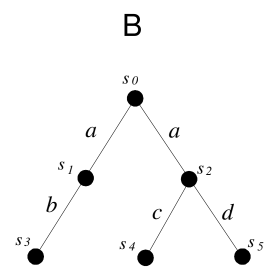
3.2 State set of a labelled transition system

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:
$G \text{ after } \mu = \{ s \in Stat \mid \exists s' \in G : s' \overset{\mu}{\Longrightarrow} s \}$. □

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

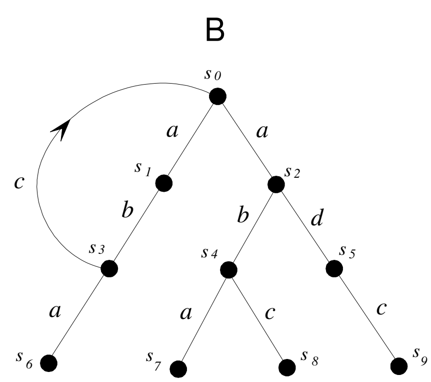
Case 2) $\underline{\sigma = \mu\sigma'}$

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 = Ø □

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)

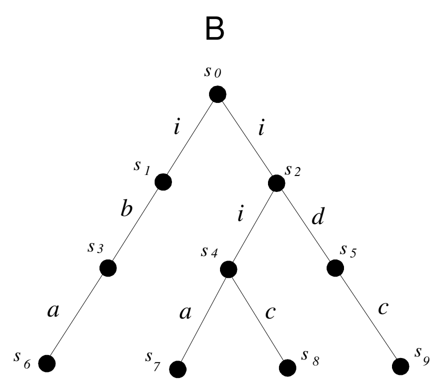
Let B = < Stat, Act, Trans, s_0 > be a LTS and G a state set of B.

Compulsory(G) = { $A \subseteq L \mid \exists g \in G$ and $s \in$ Stat : $g{=}\varepsilon{\Rightarrow}s$ and stable(s) and A = Out(s) }

Options(G) = { $a \in L \mid \exists g \in G$ and $s \in$ Stat : $g{=}\varepsilon{\Rightarrow}s$ and $\neg$ stable(s) and $s{-}a{\rightarrow}$ }

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 $B = \langle \text{Stat}, \text{Act}, \text{Trans}, s_0 \rangle$ 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:

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

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 $\text{Comp} = \text{Compulsory}(\{s\})$ and $\text{Opt} = \text{Options}(\{s\})$.

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

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 $A = \{\{a_1, a_2, \ldots a_n\}\} \cup A'$

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

3.3.5 Canonical testers and state sets

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

Let $B = \langle \text{Stat}, \text{Act}, \text{Trans}, s_0 \rangle$ 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

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

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

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

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

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

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

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

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:
$\text{Stable}(\Sigma_j\ i;B_j) = \cup_j\ \text{Stable}(B_j)$
$\text{Unstable}(\Sigma_j\ i;B_j) = \cup_j\ \text{Unstable}(B_j)$

From definition 2.6 we have:

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

$$
\begin{aligned}
& = \cup_j \text{ Compulsory}(B_j) \\
\text{Options}(B) \quad & = \{\ a \in L \mid \exists B' : B{=}\varepsilon{\Rightarrow}B'{-}i{\rightarrow} \text{ and } B'{-}a{\rightarrow}\ \} \\
& = \{\ a \in L \mid \exists B' \in \text{Unstable}(B) \text{ and } B'{-}a{\rightarrow}\ \} \\
& = \{\ a \in L \mid \exists B' \in \cup_j \text{ Unstable}(B_j) \text{ and } B'{-}a{\rightarrow}\ \} \\
& = \cup_j \{\ a \in L \mid \exists B' \in \text{Unstable}(B_j) \text{ and } B'{-}a{\rightarrow}\ \} \\
& = \cup_j \{\ a \in L \mid \exists B' : B_j{=}\varepsilon{\Rightarrow}B'{-}i{\rightarrow} \text{ and } B'{-}a{\rightarrow}\ \} \\
& = \cup_j \text{ Options}(B_j)
\end{aligned}
$$

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)	$= \cup_{g \in G}$ Compulsory({g})	
	$= \cup_{g \in G}$ <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

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

proof

$$
\begin{aligned}
\mathrm{Options}(G) &= \textstyle\bigcup_{g \in G} \mathrm{Options}(\{g\}) && \\
&= \textstyle\bigcup_{g \in G} \underline{\mathrm{Options}}(B/g) && \text{Definition of options sets} \\
&= \underline{\mathrm{Options}}(\textstyle\sum_{g \in G} i;\ B/g) && \text{Lemma 3.1} \\
&= \underline{\mathrm{Options}}(B/G) && \text{Definition of B/G}
\end{aligned}
$$

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

proof

$$
\begin{aligned}
\mathrm{Out}(G) &= \textstyle\bigcup_{g \in G} \mathrm{Out}(g) && \\
&= \textstyle\bigcup_{g \in G} \underline{\mathrm{Out}}(B/g) && \text{Definition of Out sets} \\
&= \underline{\mathrm{Out}}(\textstyle\sum_{g \in G} i;\ B/g) && \text{Definition of Out sets} \\
&= \underline{\mathrm{Out}}(B/G) && \text{Definition of B/G}
\end{aligned}
$$

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

proof

$$
\begin{aligned}
B/(G \text{ after } a) &= \textstyle\sum_{g' \in G \text{ after } a} i;\ B/g' && \\
&= \textstyle\sum_{g' \in \mathrm{Stat} \mid \exists g \in G\, :\, g = a \rightarrow g'} i;\ B/g' && \text{Definition of G after a} \\
&= \textstyle\sum_{B' \mid \exists g \in G\, :\, B/g = a \rightarrow B'} i; B' && \text{Definition of B/s} \\
&= \textstyle\sum_{g \in G} \sum_{B' \mid B/g = a \rightarrow B'} i; B' && \\
&= \textstyle\sum_{g \in G} (B/g)\ \underline{\text{after}}\ a && \text{Definition of B } \underline{\text{after}} \text{ a} \\
&= (\textstyle\sum_{g \in G} i;\ B/g)\ \underline{\text{after}}\ a &&
\end{aligned}
$$

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

- $T(G) = \underline{T}(B/G)$ (5)

proof

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

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

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

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

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

$T(G) \quad = \quad i;\ stop$

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

While:

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

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

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

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

$\underline{T}(B/G) \quad = \quad i;\ stop$

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

By using (4) we get:

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

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

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

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

$\underline{T}(B/G) \quad = \quad i;\ stop$

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

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:

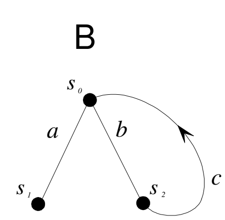
$T_{can}(B)$	$= T(\{s_0\})$	Definition of $T_{can}(B)$
	$= \underline{T}(B/\{s_0\})$	Using (5)
	$= \underline{T}(i;\ B/s_0)$	Definition of B/G
	$= \underline{T}(B/s_0)$	Property of $\underline{T}$
	$= \underline{T}(B)$	□

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 illustrate this by the following example:

Example 3.5

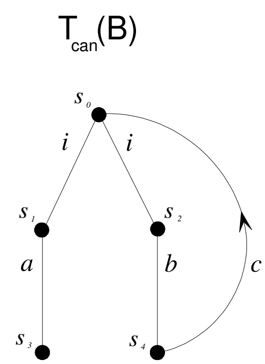
Consider the following LTS:



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′ ← Ø

    Act′ ← Act

    Trans′ ← Ø

    Visited ← Ø

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

    t_0 ← Tester($\{s_0\}$)

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

```
if ∃ tᵢ ∈ Stat′ : (G, tᵢ) ∈ Visited then
    return tᵢ
else { G is not visited yet }
    Add a new state tᵢ to Stat′
    Add (G, tᵢ) to Visited
    Co ← Compulsory(G)
    Op ← Options(G)
    if Ø ∉ Co then
        Or ← Orth(Co)
        For each set V ∈ Or do
            Add a new state tⱼ to Stat′
            Add transition < tᵢ, i, tⱼ > to Trans′
            For each action a ∈ V do
                tₖ ← Tester(G after a)
                Add transition < tⱼ, a, tₖ > to Trans′
        For each action a ∈ Op do
```

$t_j \leftarrow$ Tester(G after a)

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

else { Ø ∈ Co }

Add a new State t_j to Stat′

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

O ← Out(G)

For each action a ∈ 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 Ø ∉ Compulsory(G) then

if Options(G) = Ø and Orth(Compulsory(G)) = {V} then

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

else

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

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:

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

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).