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). $\square$

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 $\forall s : s \not- i \rightarrow$

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 $\leftarrow \emptyset$

O $\leftarrow \emptyset$

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

Add s to Visited

O $\leftarrow$ O $\cup$ { a $\in$ L | s–a→ }

For each s′ such that s–i→s′ do

if s′ $\notin$ Visited then

O $\leftarrow$ O $\cup$ 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

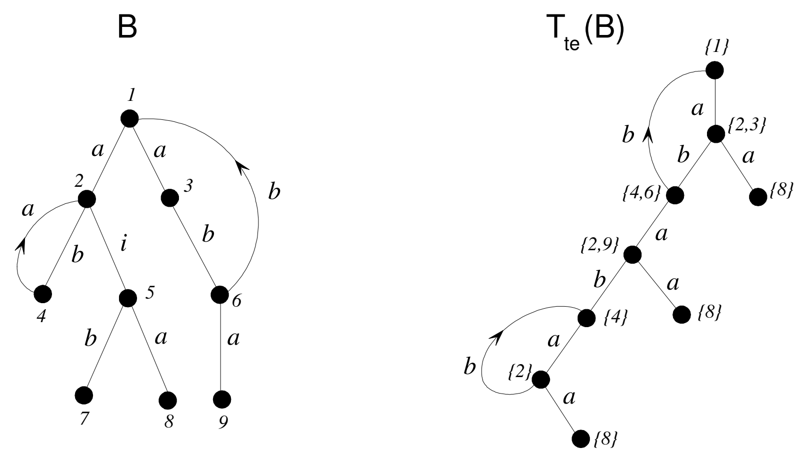
t_0 ← Tracer({s_0})

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

```
if ∃ t_i ∈ Stat′ : (G, t_i) ∈ 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 ∈ 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:
$\forall s \in Stat : Checked(s) = \{s' \in Stat' \mid Tr(s) \subseteq Tr(s')\}$

<u>Initialization</u>

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

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

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

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 $s' \in$ Checked(s) then
  return TRUE
else { $s' \notin$ Checked(s) }
  if $Out(s) \subseteq Out(s')$ then
    Add s′ to Checked(s)
    if Out (s) = Ø then
      return TRUE
    else   return $\bigwedge_{a \in Out(s) \text{ and } s-a\rightarrow t \text{ and } s'-a\rightarrow t'} Incl(t, t')$
  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

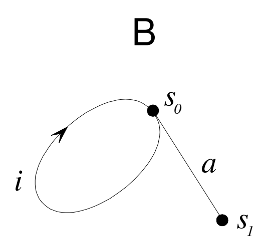
$Ref(s) =_{def} R_\varepsilon(s)$
$Ref(G) =_{def} R_\varepsilon(G)$ □

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 $Ref(s_0) = R_\varepsilon(s_0)$:

$R_\varepsilon(s_0) =_{def} \{ A \subseteq L \mid \exists s' : (s_0{=}\varepsilon{\Rightarrow}s' \text{ and } \forall\, a \in A : s'{\neq}a{\Rightarrow}) \}$

$= \{ A \subseteq L \mid \forall\, a \in A : s_0{\neq}a{\Rightarrow} \}$ 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 Ref(G) = $\bigcup_{g \in G}$ 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 ← $\bigcup_{s' \in \text{Stat} \mid s \text{–i→} s' \text{ and } s' \notin \text{Visited}}$ Ref(s′)

Remark

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

Act′ ← Act - {i}

Trans′ ← Ø

Visited ← Ø

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

t_0 ← Failure({s_0})

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

if ∃ t_i ∈ Stat′ : (G, t_i) ∈ 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) ← Ref(G)

O ← Out(G)

For each action a ∈ O do

t_j ← Failure(G after a)

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

return t_i □

Note that T_{te}(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 $B = < Stat, Act, Trans, s_0 >$ and $B' = < Stat', Act', Trans', s_0' >$:

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 $Act \cup Act'$ 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 $F = < Stat, Act, Trans, R, s_0 >$ and $F' = < Stat', Act', Trans', R', s_0' >$.
To each state s of F we associate a set of states of F′ : $Checked(s) \subseteq Stat'$.

<u>Initialization</u>

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

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

B conf B′ $\Leftrightarrow$ $Conf(s_0, s_0')$

Where in general, given $s \in Stat$ and $s' \in Stat'$, 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′) | A $\subseteq$ A′ then
        Add s′ to Checked(s)
        if Out(s) $\cap$ Out(s′) = Ø then
            return TRUE
        else   return $\bigwedge_{a \in \text{Out}(s) \cap \text{Out}(s') \text{ and } s-a\rightarrow t \text{ and } s'-a\rightarrow t'}$ 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

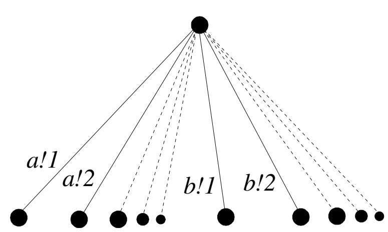
Like the semantics of Basic LOTOS, the semantics of Full LOTOS can also be defined in terms of labelled transition systems. However, simple Full LOTOS behaviour expressions may be defined with an initial behaviour that is infinitely branching. In this case the algorithm implementing the CO-OP method 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

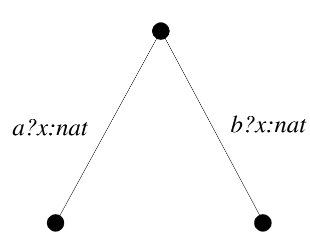
Consider the following Full LOTOS behaviour expression:

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 section 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.

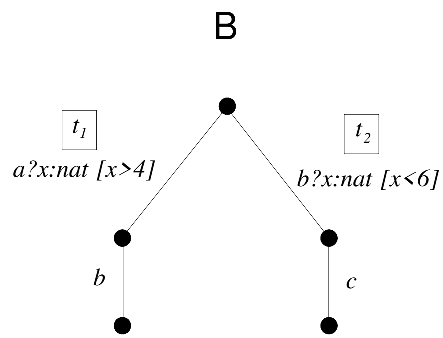
5.2.1 Overlapping transitions

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):

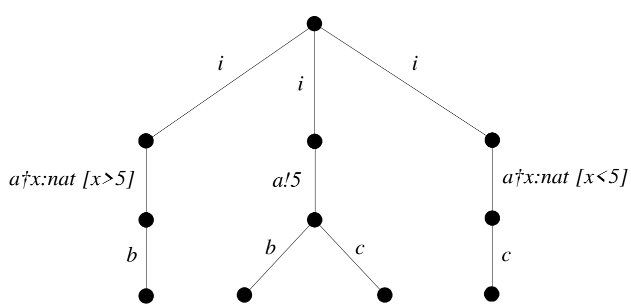
T = a!5; b; stop

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:

T(B)



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:

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

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

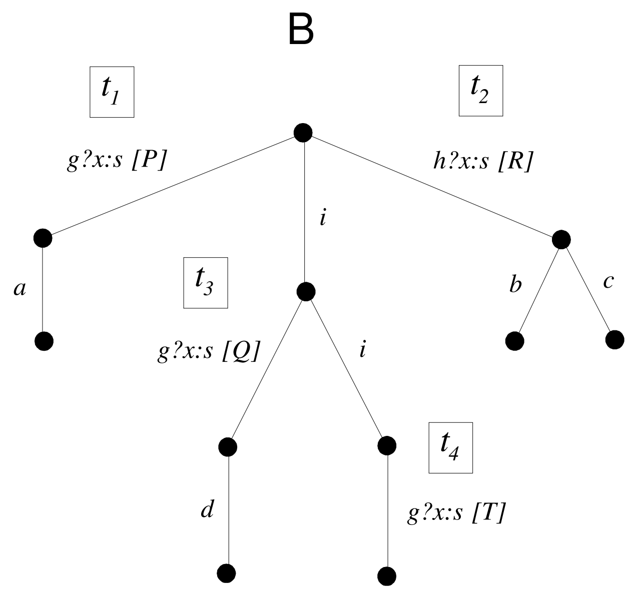
Suppose we have *n* overlapping transitions, then each transition t_i will be replaced by 2^{n-1} (number of possible Q_r's) new transitions. This gives an idea about the size of the

[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:



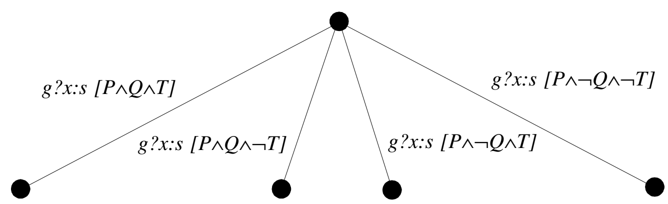
We assume that transitions t_1, t_3 and t_4 may overlap since they are name-sort identical and belong to the same *Out* set of the initial state of the pLTS B.

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.

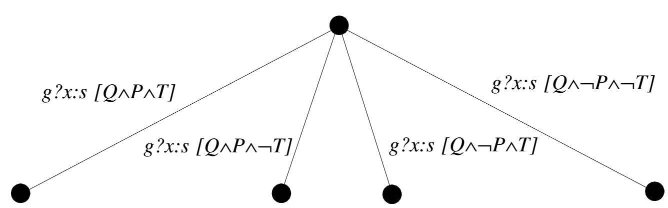
Each transition of $\{t_1, t_3, t_4\}$ will be replaced by $4 = 2^{3-1}$ new transitions.

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

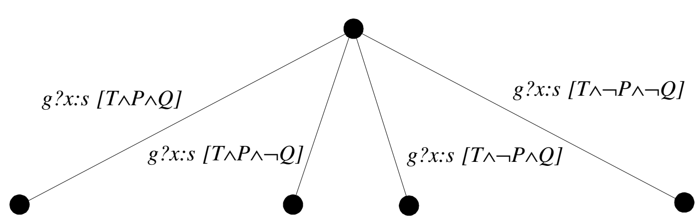
Transition t_1 is replaced by the following transitions:



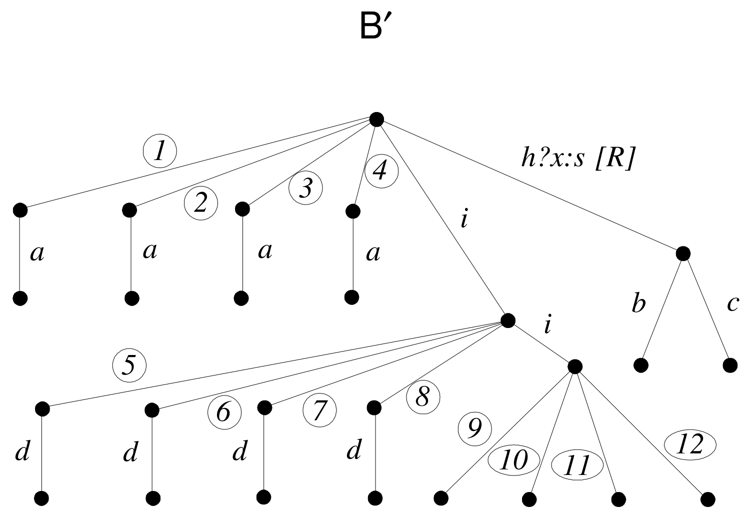
Transition t_3 is replaced by the following transitions:



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:

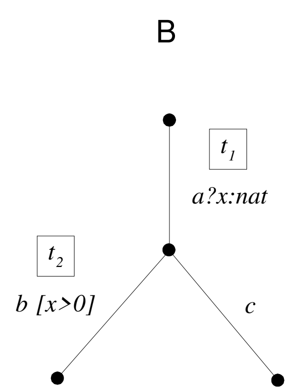
(i) The predicate depends on an already bound variable and is not true.

(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).

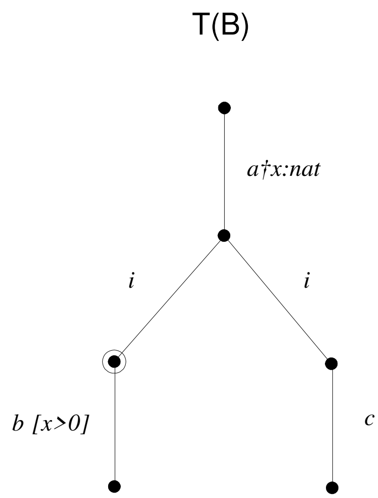
Example 5.3

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:

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

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

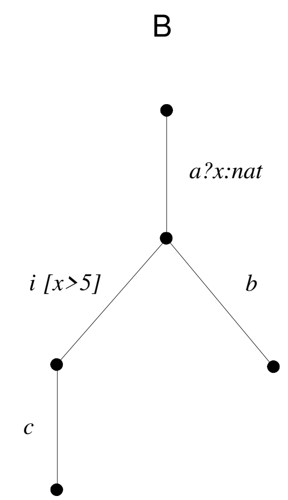
Computing the set DL may require solving a set of mathematical equations that can be very complex.

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:

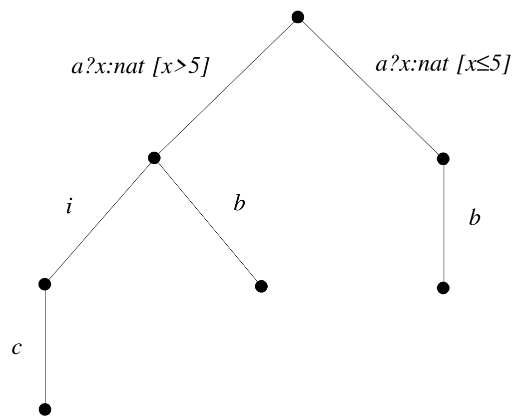


Action *b* is compulsory or optional depending on the value of the variable *x*.

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.

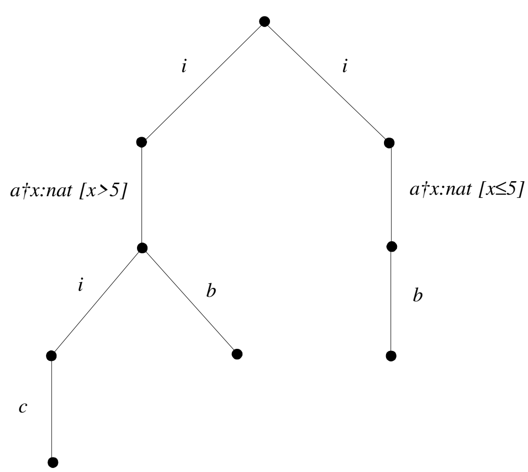
B is testing equivalent to the following pLTS:

B′



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

T(B) = T(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:

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

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:

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

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.

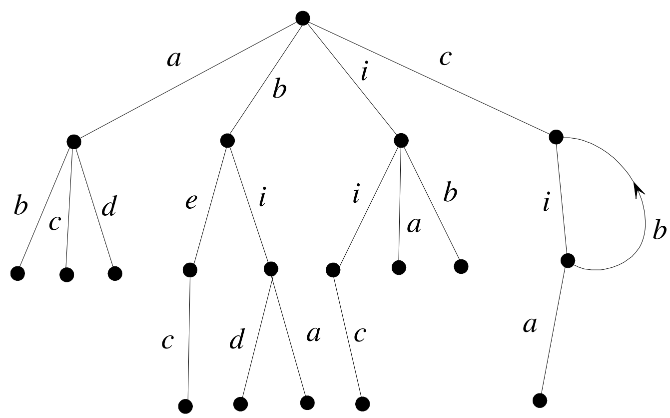
Remark

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.

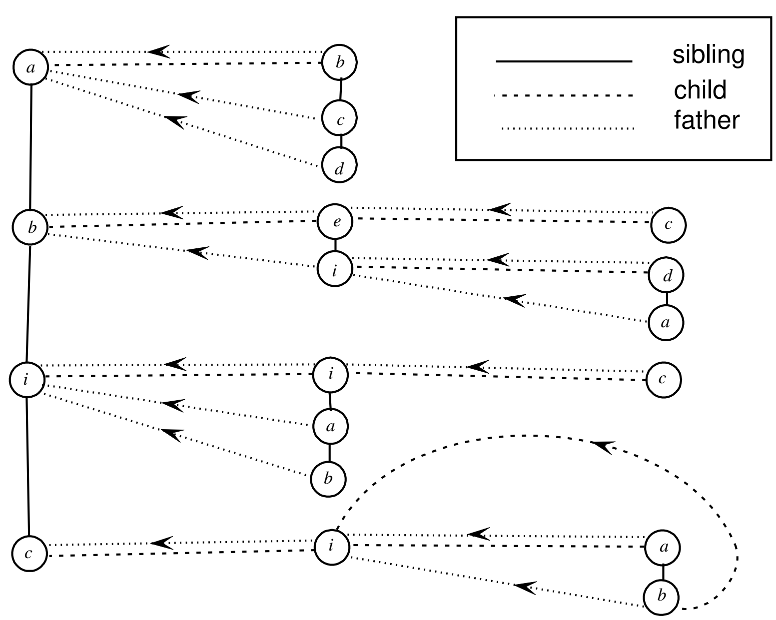
The following example shows a symbolic execution tree and how it is stored in memory by LOTEST.

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

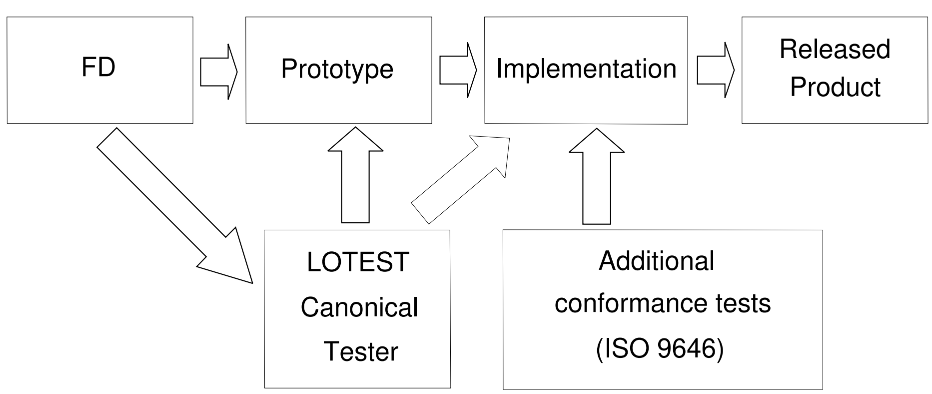
In this thesis, we have summarized the basic theory of conformance testing for LOTOS based applications, we have proposed some generalizations of this theory, and we have described a tool that we have implemented in order to demonstrate the theory and its generalization.

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 ***t****tname1* 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 ***p****tname1*.

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 ***r****tname1*. 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).

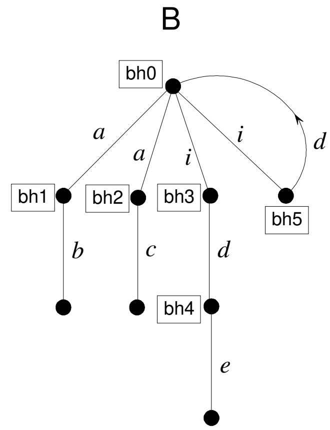
QUIT

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:

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

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 "†" 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).