Chapter 4

The Model Checking Algorithms

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

4.1 The Checking Algorithms

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

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

□

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

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

4.2.1 The Next Time Operators

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

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

□

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

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

□

4.2.2 The Global Operators

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

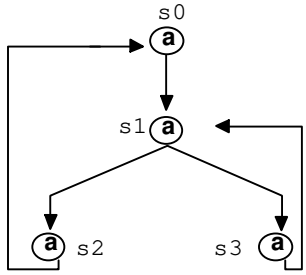
```
 1.  if  visited(s_i) then
 2.              holds(AG(f),s_i) ← true
 3.              return
 4.  else
 5.              visited(s_i) ← true
 6.              if  holds(f,s_i)  then
 7.                      for all successors t of s_i such that (s_i,t) ∈ ℜ do
 8.                              if  ¬ holds(AG(f),t)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

□

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

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

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



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

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

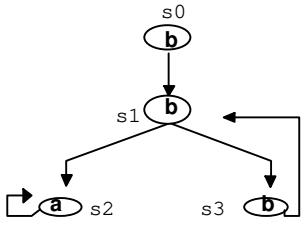
```
1.  if visited(s_i) then
2.          holds(EG(f),s_i) ← true
3.          return
4.  else
5.          visited(s_i) ← true
6.          if holds(f,s_i) then
7.                  for all successors t of s_i such that (s_i,t) ∈ ℜ do
8.                          if holds(EG(f),t) then
9.                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

□

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

Example

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



□

4.2.3 The Future Operators

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

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

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

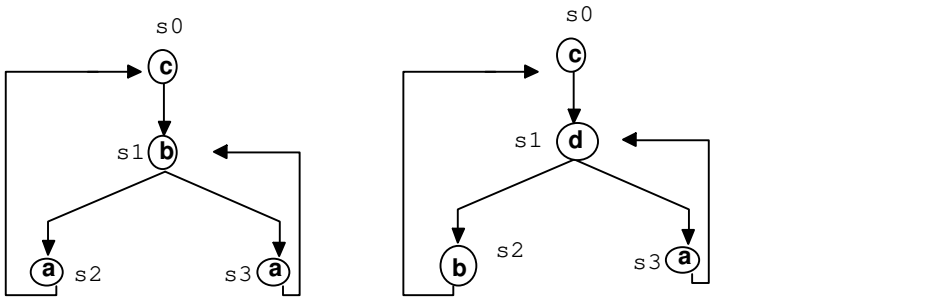
Similarily, we can derive the algorithm for *EF(f)* as follows:

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

□

Example

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



□

4.2.4 The Until Operators

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

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

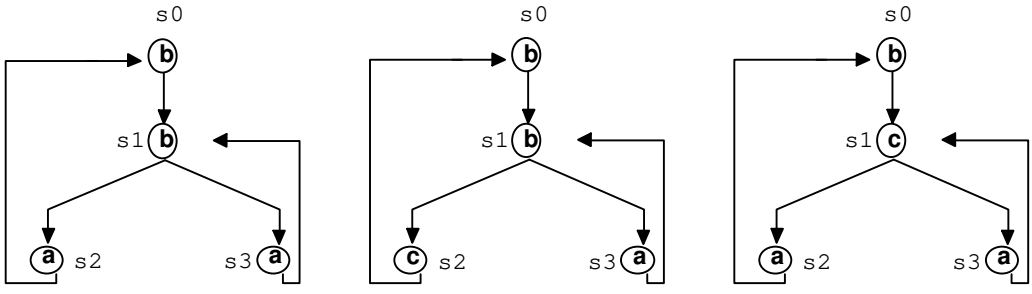
For $A[f_1 \upsilon f_2]$, the algorithm returns false when a visited state is reached again. This implies that a cycle has been detected on which f_1 holds but f_2 is never reached (lines 1 to 3). It should be noticed that if f_2 holds at a particular state, then $A[f_1 \upsilon f_2]$ is *true* at that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

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

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

□

Example



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

□

4.2 Complexity of The Algorithms

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

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

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

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

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

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

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

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

4.3 Adaptation to LOTOS

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

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

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

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

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

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

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

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

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

4.4 A Modified Satisfaction Relation

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

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

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

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

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

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

□

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

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

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

□

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

4.5 The Modified Algorithms

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

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

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

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

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

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

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

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

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

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

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

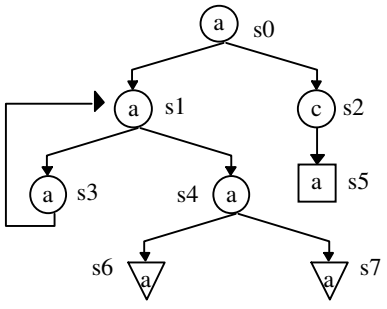
□

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

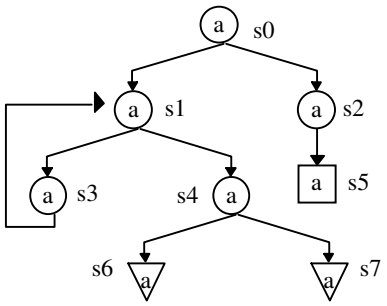
Example

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

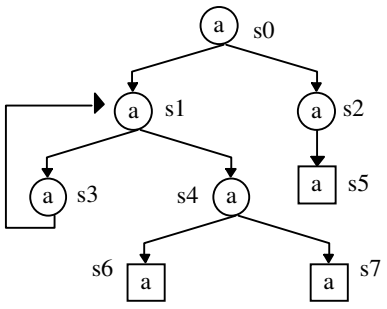
found to be false at state s_2 which is enough to conclude that *AG(a)* will not be satisfied even if we increase the depth of the tree.



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



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



□

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

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

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

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

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

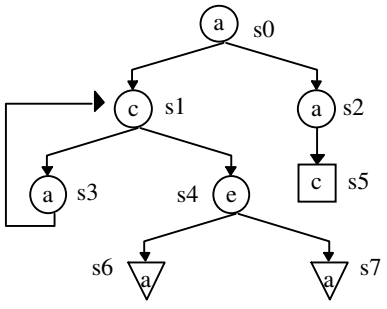
```
1.  if visited(s_i) then
2.          holds(E[f_1 υ f_2],s_i) ← false
3.          return
4.  else
5.          visited(s_i) ← true
6.          if holds(f_2,s_i) then
7.                  holds(E[f_1 υ f_2],s_i) ← true
8.                  return
9.          else
10.                 if ¬ holds(f_1,s_i)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

□

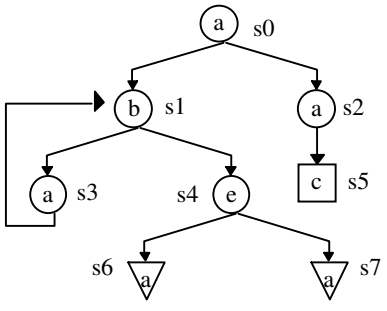
Example

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

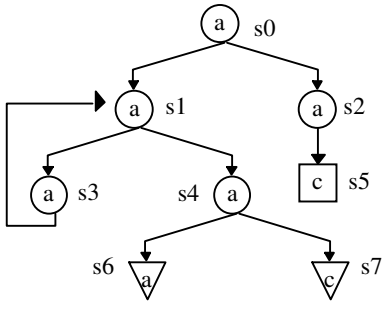
because the tree is cut and we need to increase the depth of the tree to make such conclusion.



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



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



□

4.6 Chapter Summary

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

Chapter 5

LMC Design and Implementation

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

5.1 General Structure of LMC

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

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

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

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

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

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

□

LMC consists of three main modules:

(i) the interface module

(ii) the transformer

(iii) the model checker

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

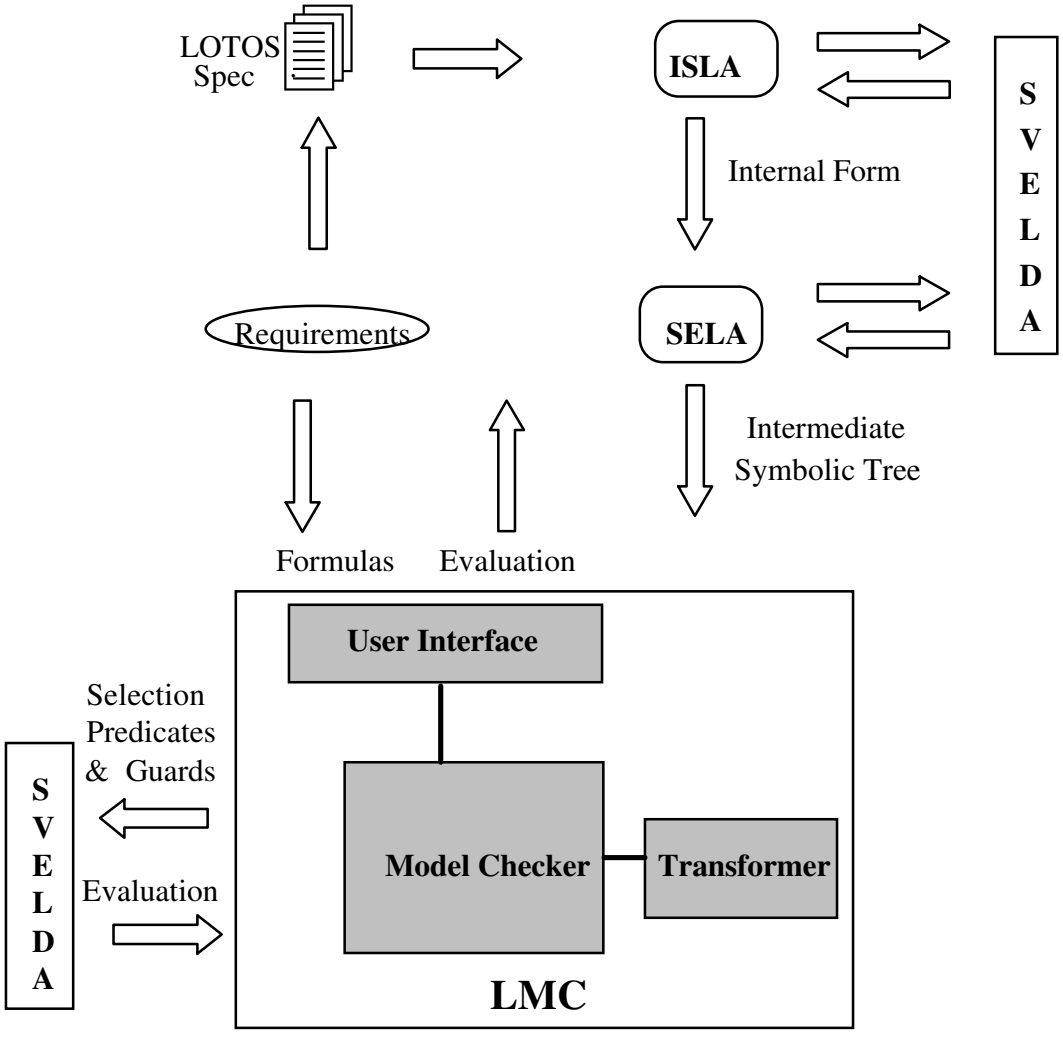


Figure 5.1 Overall Structure of LMC

5.2 The Interface Module

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

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

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

5.3 The Transformer

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

5.3.1 The Tree Model

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

(i) An initial state.

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

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

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

(iii) A transition relation between the states.

□

5.3.2 An Example

Consider the LOTOS behaviour given by:

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

SELA generates the following intermediate symbolic behaviour tree:

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

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

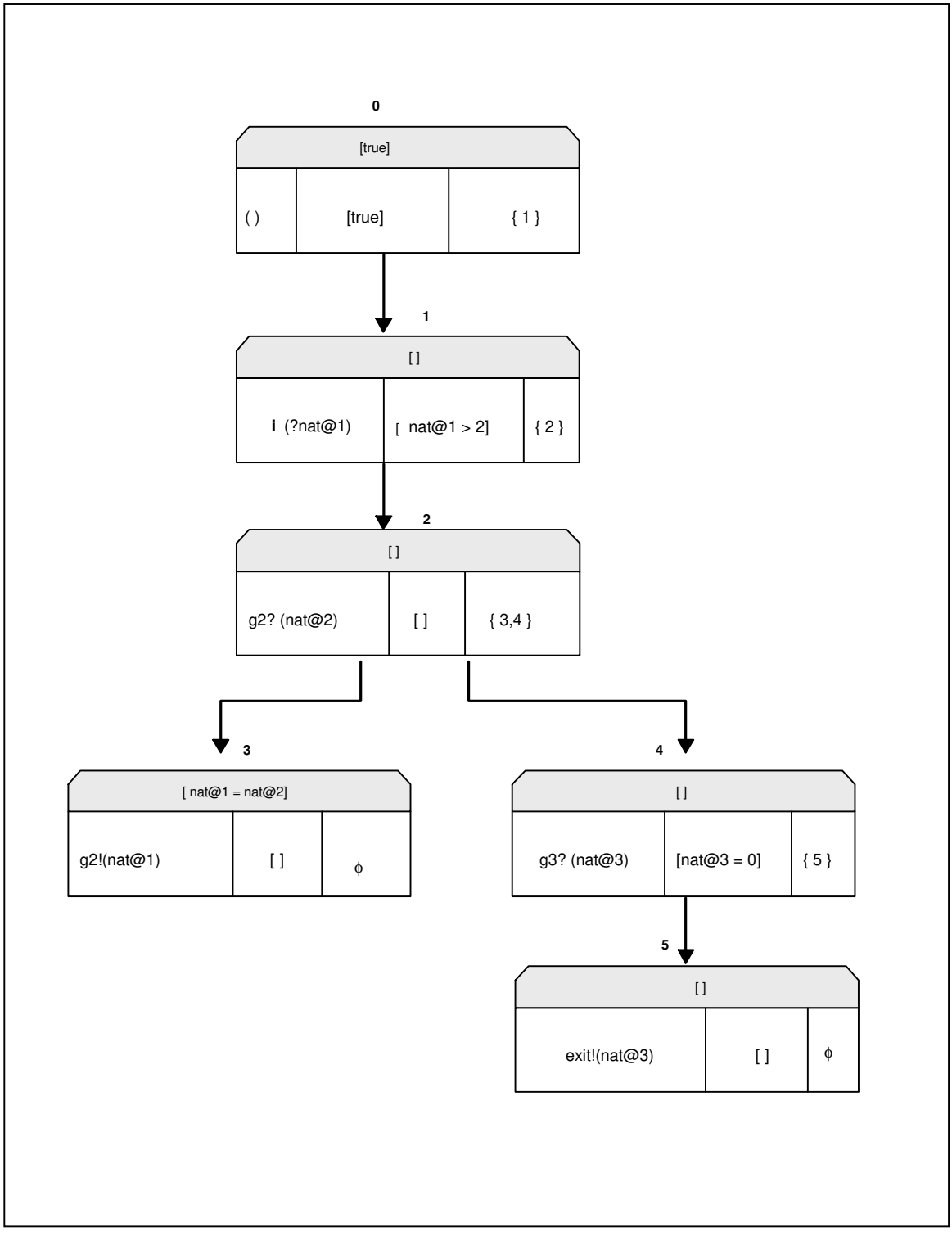


Figure 5.2 tree model used by the model checker

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

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

Table 5.1 Components of the states

It is important to notice the following aspects:

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

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

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

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

□

5.4 The Model Checker

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

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

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

5.4.1 Formulation of the Properties

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

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

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

where:

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

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

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

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

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

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

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

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

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

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

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

5.4.2 The Synchronization Procedure

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

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

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

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

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

Example

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

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

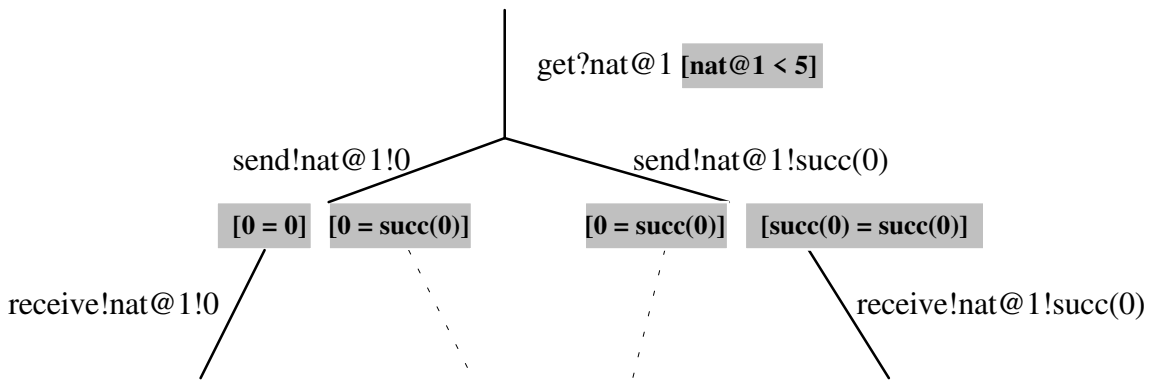
Synchronization is impossible in case *2* since the value *7* does not satisfy the restricting condition. In case 5, the restricting condition is neglected since the value offered is a "*". In case 8, synchronization occurs only if *nat@1* was previously assigned the value *0*.

5.4.3 Evaluation of Selection Predicates and Guards

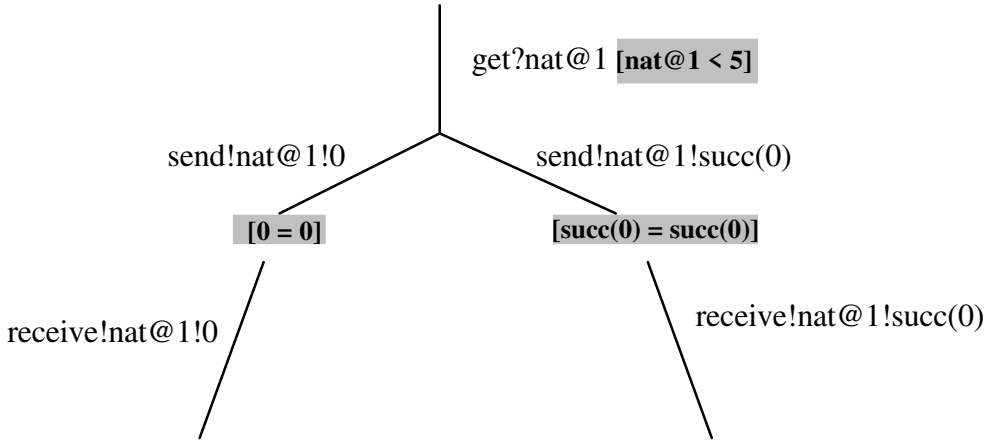
During the model checking procedure, variables in the symbolic tree are instantiated due to the synchronization with the actions of the formulas. Therefore, various execution sequences in the tree can be explored in order to determine the satisfiability of the formulas. During the exploration of such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

Example

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



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



□

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

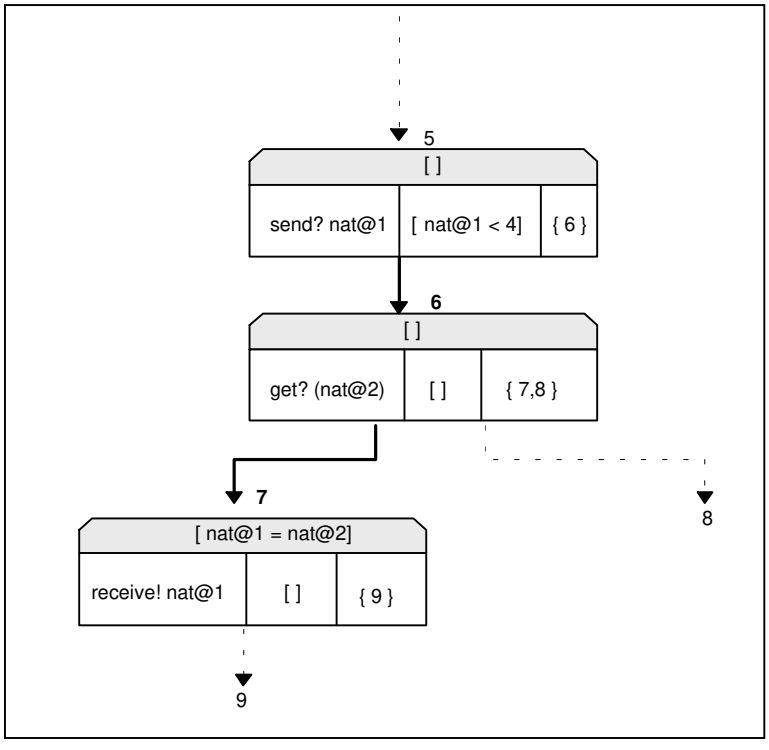
Example

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

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

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

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



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

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

□

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

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

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

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

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

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

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

5.5 Properties Handled by LMC

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

The correctness properties that we address include:

Absence of deadlock

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

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

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

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

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

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

$$AG(\neg deadlock).$$

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

$$AG(a \rightarrow AF(deadlock)).$$

Absence of starvation

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

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

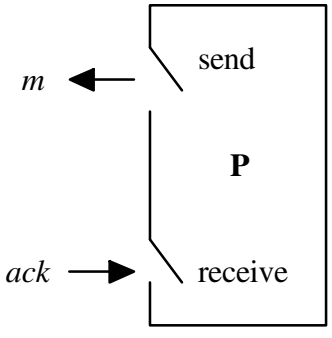


Figure 5.3 Process P

Precedence properties

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

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

□

5.6 Diagnostics

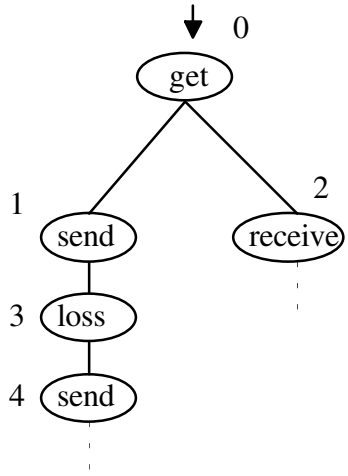
A model checker usually verifies whether properties expressed in temporal logic are satisfied on a given model. However, it does not explain why a certain formula fails or succeeds. For this reason, appropriate diagnostics must accompany a model checker in order to help the user detect erratic behaviours.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

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

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

Example

Given the following simple model:



Suppose that we want to prove the following formula:

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

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

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

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

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

Chapter 6

Case Studies

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

6.1 Example 1: A Data Link Service Provider

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

6.1.1 Informal Description of the Service Provider

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

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

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

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

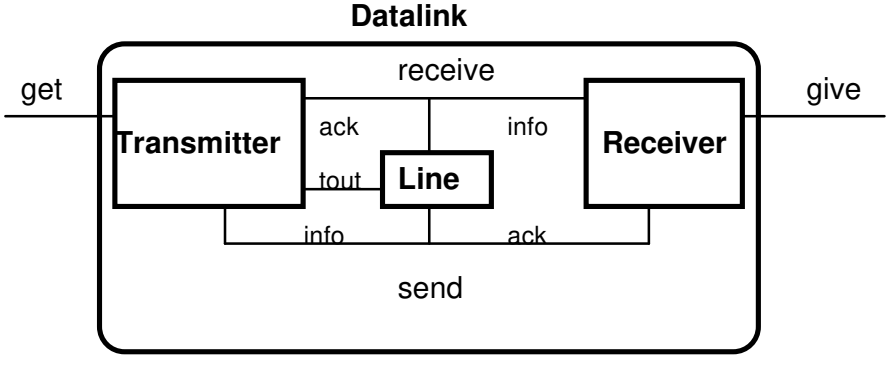


Figure 6.1 The Data Link Service Provider

6.1.2 The LOTOS Specification

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

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

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

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

6.1.3 The Symbolic Tree

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

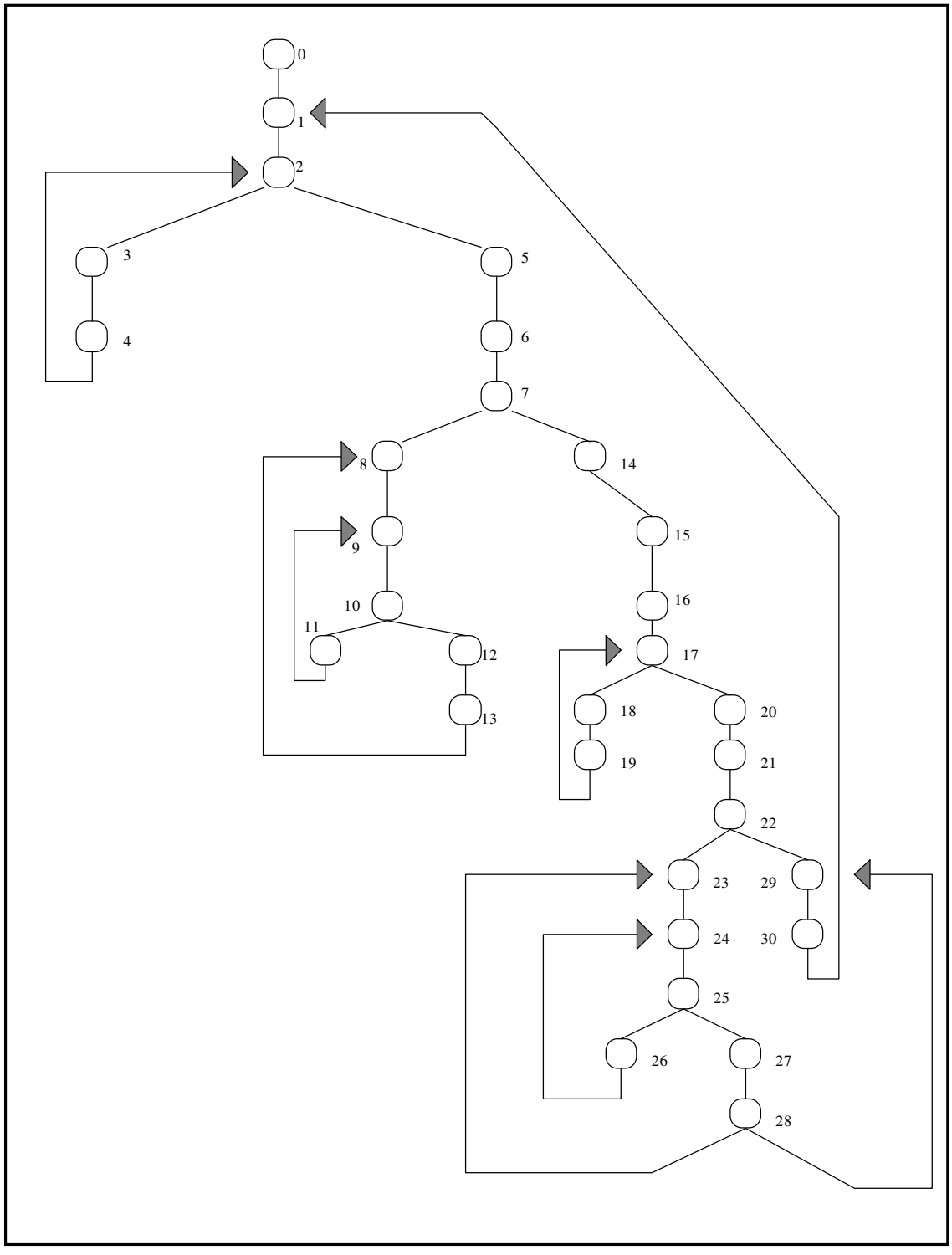


Figure 6.2 Symbolic tree for the datalink service provider

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

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

6.1.4 Verification Phase

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

1) Absence of deadlock.

$$AG(\neg deadlock)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2 Example 2: A Transport Service Handler

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

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

6.2.1 Informal Description of the Service

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

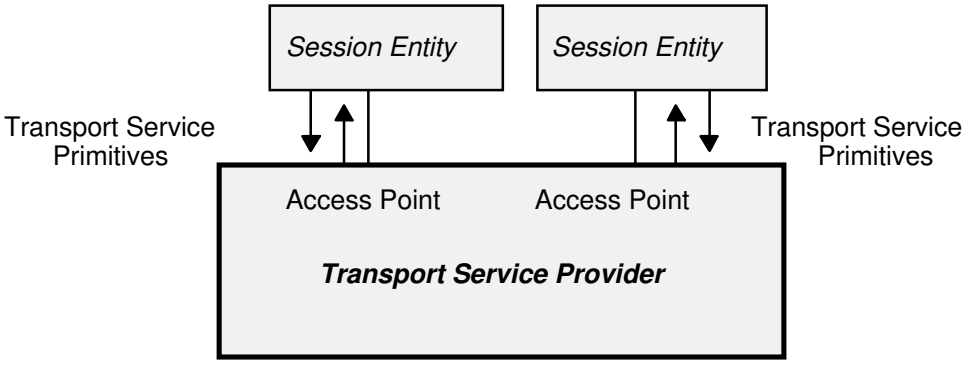


Figure 6.3 Transport Service

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

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

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

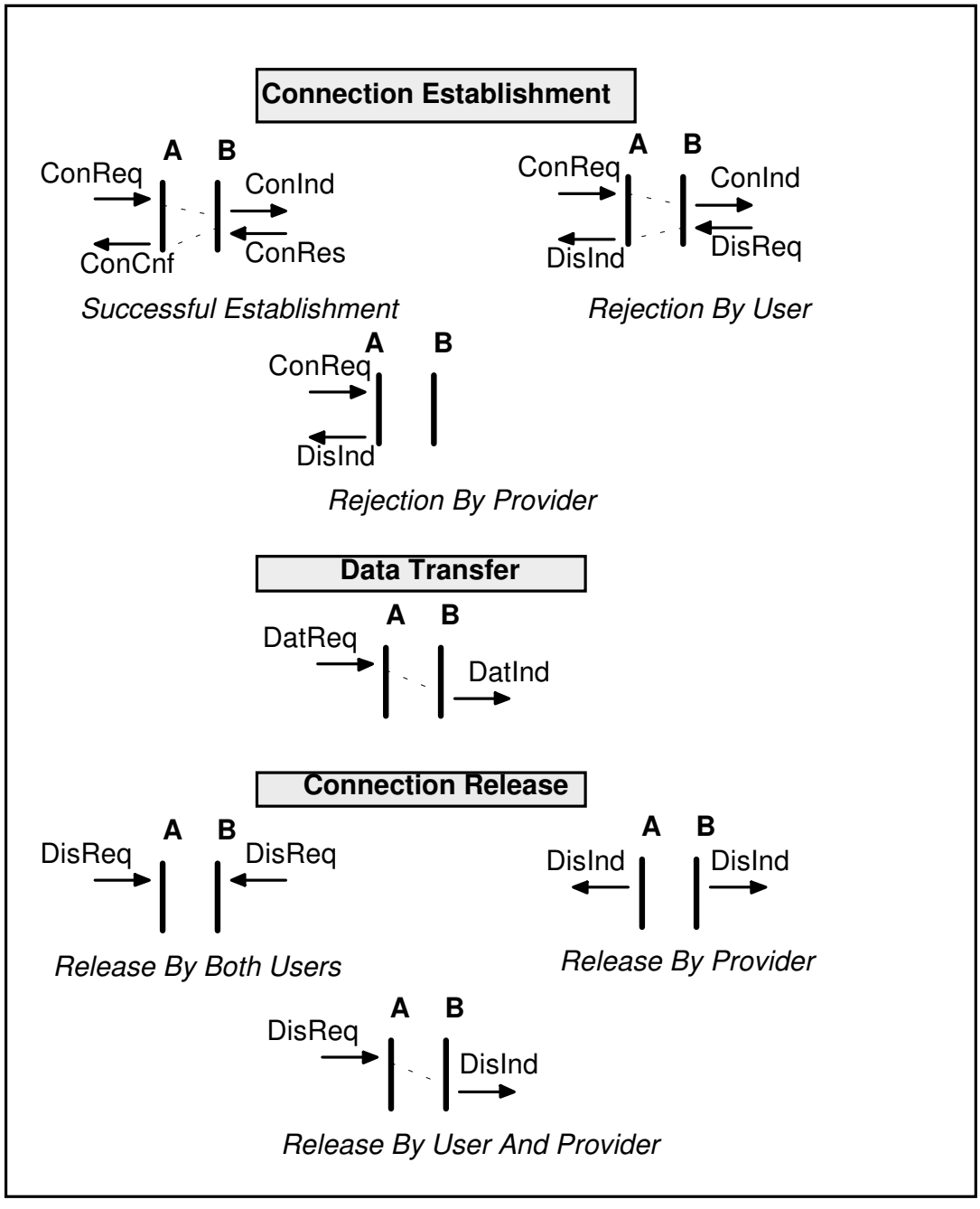


Figure 6.4. Transport Service Primitives

6.2.2 The LOTOS Specification

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

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

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

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

where process *Handler* is defined as follows:

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

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

6.2.3 The Symbolic Tree

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

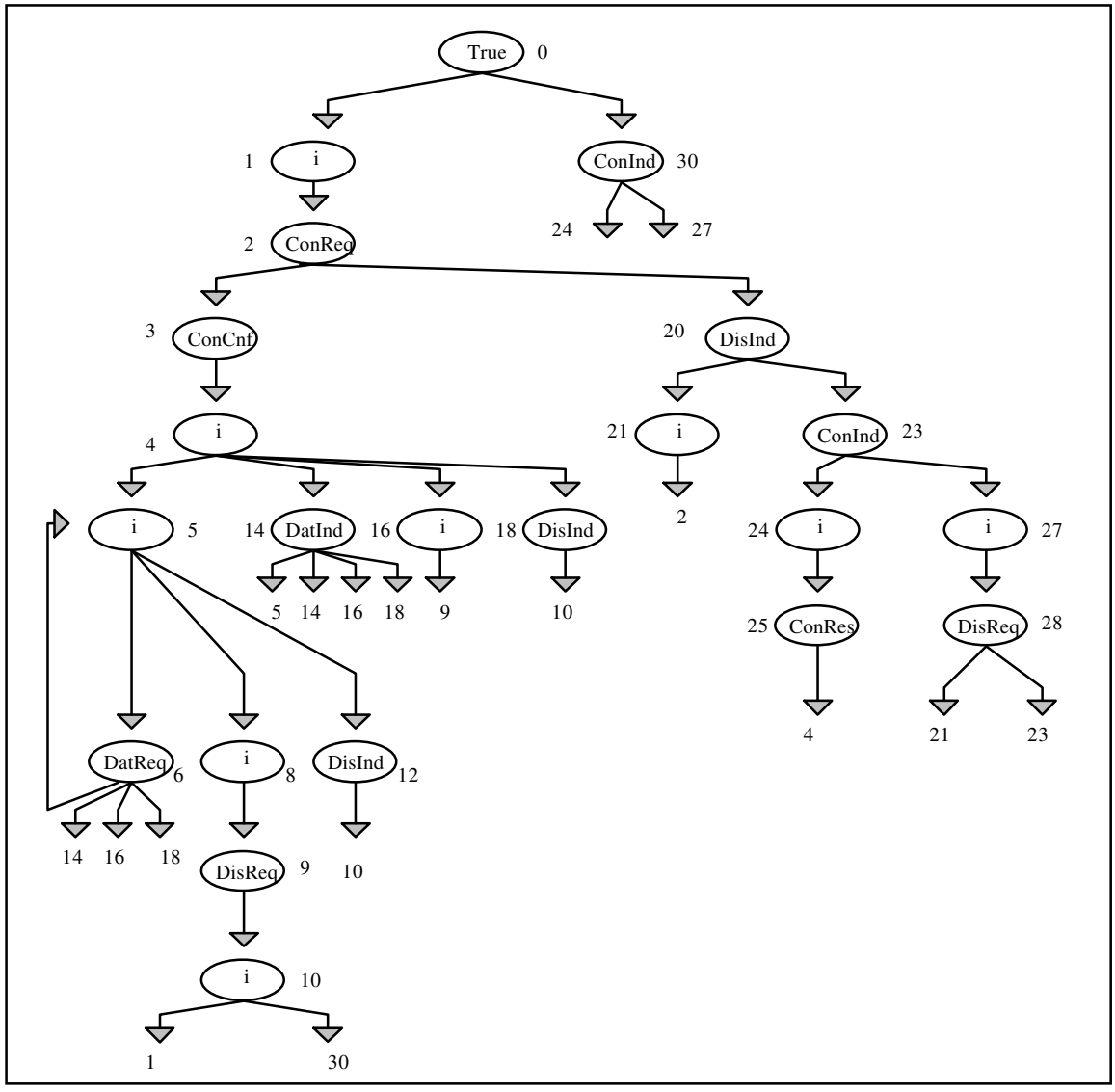


Figure 6. 5 Symbolic tree for the Transport Service Handler

6.2.4 Verification Phase

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

1) Absence of deadlock.

$$AG(\neg deadlock)$$

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

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

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

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

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

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

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

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

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

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

Chapter 7

Conclusions and Future Work

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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