Temporal Debugging for Concurrent Systems

Elsa Gunter[1] and Doron Peled[2]

[1] Department of Computer Science
New Jersey Institute of Technology
Newark, NJ 07102, USA
[2] Dept. of Elect. and Computer Eng.
The University of Texas at Austin
Austin, TX 78712, USA

Abstract. Temporal logic is often used as the specification formalism for the automatic verification of finite state systems. The automatic temporal verification of a system is a procedure that returns a yes/no answer, and in the latter case also provides a counterexample. In this paper we suggest a new application for temporal logic, as a way of assisting the debugging of a concurrent or a sequential program. We employ temporal logic over finite sequences as a constraint formalism that is used to control the way we step through the states of the debugged system. Using such temporal specification and various search strategies, we are able to traverse the executions of the system and obtain important intuitive information about its behaviors. We describe an implementation of these ideas as a debugging tool.

1 Introduction

Temporal logic is a specification formalism that is often used to express properties of software and hardware systems. Model checking techniques allow us to check a finite state description of a system against its temporal specification, and provide a counter example in case the property does not hold.

In this paper we suggest to extend the use of a temporal specification, and use temporal logic for interactively controlling the debugging of systems. We allow specifying temporal properties of *finite sequences*. A debugger is enriched with the ability to progress from one step to another via a finite sequence of states that satisfy a temporal property.

The usual mode of debugging involves stepping through the states of a system (program) by executing one or several transitions (with different granularities, e.g., a transition can involve the the execution of a procedure). Debugging concurrent systems is harder, since there are several cooperating processes that need to be monitored. Stepping through the different transitions can be applied in many different ways. Instead, we allow applying a temporal property that describes a finite sequence of concurrent events that need to be executed from the current state, leaping into the next state.

We interpret linear temporal logic (LTL) on finite sequences. The automatic translation from LTL to finite state automata in [4] is adapted to include the finite case. We describe various search algorithms that can be used for generating appropriate paths and states during a debugging session.

J.-P. Katoen and P. Stevens (Eds.): TACAS 2002, LNCS 2280, pp. 431–444, 2002.

2 Defining LTL on Finite Sequences

One of the most popular specification formalisms for concurrent and reactive systems is Linear Temporal Logic (LTL) [7]. Its syntax is as follows:

$$\varphi ::= (\varphi) \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \overline{\bigcirc}\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \varphi \,\mathcal{V}\, \varphi \mid p$$

where $p \in \mathcal{P}$, with $\mathcal{P}$ a set of propositional letters. We denote a propositional sequence over $2^{\mathcal{P}}$ by σ, its ith state (where the first state is numbered 0) by $\sigma(i)$, and its suffix starting from the ith state by $\sigma^{(i)}$. Let $|\sigma|$ be the length of the sequence Σ, which is a natural number. The semantic interpretation of LTL is as follows:

- $\sigma \models \bigcirc\varphi$ iff $|\sigma| > 1$ and $\sigma^{(1)} \models \varphi$.
- $\sigma \models \varphi \,\mathcal{U}\, \psi$ iff $\sigma^{(j)} \models \psi$ for some $0 \leq j < |\sigma|$ so that for each $0 \leq i < j$, $\sigma^{(i)} \models \varphi$.
- $\sigma \models \neg\varphi$ iff it is not the case that $\sigma \models \varphi$.
- $\sigma \models \varphi \vee \psi$ iff either $\sigma \models \varphi$ or $\sigma \models \psi$.
- $\sigma \models p$ iff $|\sigma| > 0$ and $\sigma(0) \models p$.

The rest of the operators can be defined using the above operators. In particular, $\overline{\bigcirc}\varphi = \neg\bigcirc\neg\varphi$, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi \,\mathcal{V}\, \psi = \neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$, $\mathit{true} = p \vee \neg p$, $\mathit{false} = p \wedge \neg p$, $\Box\varphi = \mathit{false}\,\mathcal{V}\,\varphi$, and $\Diamond\varphi = \mathit{true}\,\mathcal{U}\,\varphi$. The operator $\overline{\bigcirc}$ is a 'weak' version of the $\bigcirc$ operator. Whereas $\bigcirc\varphi$ means that φ holds in the suffix of the sequence starting from the next state, $\overline{\bigcirc}\varphi$ means that *if* the current state is not the last one in the sequence, *then* the suffix starting from the next state satisfies φ.

We distinguish between the operator $\bigcirc$, which we call *strong nexttime*, and $\overline{\bigcirc}$, which we call *weak nexttime*. Notice that

$$(\bigcirc\varphi) \wedge (\overline{\bigcirc}\psi) = \bigcirc(\varphi \wedge \psi), \tag{1}$$

since $\bigcirc\varphi$ already requires that there will be a next state. Another interesting observation is that the formula $\overline{\bigcirc}\mathit{false}$ holds in a state that is in deadlock or termination.

The operators $\mathcal{U}$ and $\mathcal{V}$ can be characterized using a recursive equation, which is useful for understanding the transformation algorithm, presented in the next section. Accordingly, $\varphi \,\mathcal{U}\, \psi = \psi \vee (\varphi \wedge \bigcirc\varphi \,\mathcal{U}\, \psi)$ and $\varphi \,\mathcal{V}\, \psi = \psi \wedge (\varphi \vee \overline{\bigcirc}(\varphi \,\mathcal{V}\, \psi))$.

3 Finite LTL Translation Algorithm

We modify the algorithm presented in [4] for translating an LTL formula φ into an automaton $\mathcal{B} = \langle S, I, \delta, F, D, L\rangle$, where S is a set of *states*, $I \subseteq S$ is a set of *initial states*, $\delta \subseteq S \times S$ is the *transition relation*, $F \subseteq S$ are the *accepting states*, D is a set of *state labels*, and $L : S \to D$ is the *labeling function*. Note that $\mathcal{B}$ is an automaton on finite words, unlike a Büchi automaton, which is usually resulted

from translating LTL formulae over infinite sequences, and which recognizes infinite sequences.

As a preparatory step, we bring the formula φ into *negation normal form* as follows. First, we push negation inwards, so that only propositional variables can appear negated. To do that, we use LTL equivalences, such as $\neg\Diamond\psi = \Box\neg\psi$. One problem is that pushing negations into until ($\mathcal{U}$) subformulas can explode the size of the formula. To avoid that, we use the operator *release* ($\mathcal{V}$), which is the dual of the operator *until*. We also remove the eventuality ($\Diamond$) and always ($\Box$) operators using the *until* and *release* operators and the equivalences $\Diamond\psi = \mathit{true}\ \mathcal{U}\ \psi$ and $\Box\psi = \mathit{false}\ \mathcal{V}\ \psi$ as mentioned before.

The algorithm uses the following fields for every generated node of $\mathcal{B}$:

id A unique identifier of the node.
incoming The set of edges that are pointed into the node.
new A set of subformulas of the translated formula, which need to hold from the current node and have not yet been processed.
old A set of subformulas as above, which have been processed.
next A set of subformulas of the translated formula, which have to hold for every successor of the current node.
strong A flag that signals whether the current state must not be the last one in the sequence.

The algorithm starts with a single node, having one incoming edge from a dummy node called *init*. Its field *new* includes the translated formula φ in the above normal form, and the fields *old* and *next* are empty. A list *completed-nodes* is initialized as empty. The algorithm proceeds recursively: for a node x not yet in *completed-nodes*, it moves a subformula η from *new* to *old*. The algorithm then splits the node x into left and right copies while adding subformulas to the fields *new* and *next* according to the following table. The fields *old* and *incoming* retain their previous values in both copies, while the field *strong* can be updated if needed. The algorithm continues recursively with the split copies.

The following split table shows the new values added to the fields *new* and *next* in the left or right copies. The column 'set *strong*' indicates when the current state cannot be the last one in the sequence, namely the formulas in the *next* field are upgraded from weak nexttime to strong nexttime. (There is no need for 'set *strong*' for the right copy, only the left one.) It is sufficient to use the *strong* field rather than keeping two separate fields, for the weak and for the strong nexttime requirements, because of Equation (1).

Formula	left *new*	left *next*	set *strong* left	right *new*	right *next*
$\mu\,\mathcal{U}\,\eta$	$\{\mu\}$	$\{\mu\,\mathcal{U}\,\eta\}$	$\surd$	$\{\eta\}$	$\emptyset$
$\mu\,\mathcal{V}\,\eta$	$\{\eta\}$	$\{\mu\,\mathcal{V}\,\eta\}$		$\{\mu,\eta\}$	$\emptyset$
$\mu \vee \eta$	$\{\mu\}$	$\emptyset$		$\{\eta\}$	$\emptyset$
$\mu \wedge \eta$	$\{\mu,\eta\}$	$\emptyset$		–	–
$\overline{\bigcirc}\mu$	$\emptyset$	μ		–	–
$\bigcirc\mu$	$\emptyset$	μ	$\surd$	–	–
p, $\neg p$	$\emptyset$	$\emptyset$		–	–

When there are no more subformulas in the field *new* of the current node x, x is compared against the nodes in the list *completed-nodes*. If there is a node y that agrees with x on the fields *old* and *next*, one adds to the field *incoming* of y the incoming edges of x (hence, one may arrive to the node y from new directions). Otherwise, one adds x to that list and a new node is initiated as follows:

(a) *id* contains a new value,
(b) the *incoming* field contains an edge from x,
(c) the *new* field contains the set of the subformulas in the *next* field of x,
(d) the fields *old* and *next* are empty, and
(e) the field *strong* is initially set to *false*.

After the above algorithm terminates, we can construct the component of the automaton $\mathcal{B}$ for the translated automaton φ as follows. The states S are the nodes in *completed-nodes*. Let $P \subseteq \mathcal{P}$ be the set of propositional letters that appear in the formula φ. The set of labels D are the conjunctions of propositions and negated propositions from P (thus, there are 3^P labels in D, since each proposition may appear, not appear, or appear negated). In the constructed automaton, each node $x \in S$ is *labeled* by the propositions and negated propositions in its field *old*. The initial nodes I are those which have an incoming edge from the dummy node *init*. The transition relation δ includes pairs of nodes (s, s') if s belongs to the field *incoming* of s'. The accepting (final) states satisfy the following:

- For each subformula of φ of the form $\mu \,\mathcal{U}\, \eta$, either the *old* field contains the subformula η, or does not contain $\mu \,\mathcal{U}\, \eta$.
- the *strong* bit is set to *false*.

We may check that there is at least one path from a state in I to a state in F. Otherwise, the automaton does not accept any sequence (and no sequence is accepted by the formula φ).

We denote the system automaton by $\mathcal{A} = \langle X, J, \Delta, E, G\rangle$, where X are the states, $J \subseteq X$ are the initial states, $\Delta \subseteq X \times X$ is the transition relation, $E = 2^{\mathcal{P}}$ are the set of labels, and $G : X \to E$ is the labeling function. Each label $l \in E$ is a subset of the set of propositions $\mathcal{P}$. We can view l as a conjunction, where a proposition $p \in \mathcal{P}$ appears nonnegated if $p \in l$, and negated otherwise. Note that the labels D of $\mathcal{B}$ also allow a proposition not to appear. This allows us to combine several assignments to the propositions into one property automaton state. In the system automaton, this is not necessary, and each system state should induce a truth assignment to all the propositions. For the system automaton there are no accepting states (or we can view it as if all the states in X are accepting).

The set of propositional letters $\mathcal{P}$ available is determined by the variables in the checked program and the set of labeled on nodes in the (automatically generated) flow graphs of its processes. For a process P_i in the program, and a node l in the flow graph for that process, we can have a propositional letter Pi_at_l. In addition, we can have atoms that correspond to comparisons, e.g., for

a variable v occurring in the program, and a possible constant value x, we can have propositions representing the comparisons $v = x$, $v < x$, $v > x$, $v \leq x$, and $v \geq x$. In an automaton $\mathcal{A}$ that models the program, each propositional letter obtains the correct truth value in each state. For example, if $p \in \mathcal{P}$ corresponds to $v \leq 3$, then p will belong to (or will hold in) exactly all the states where v is less than or equals 3.

The automata product $\mathcal{B} \times \mathcal{A}$ has the following components:

- The states R are the elements of $S \times X$ where the components have compatible labeling, namely, $\{(s, x) | G(x) \rightarrow L(s)\}$. Note that $G(x) \rightarrow L(s)$ means that the assignment $G(x)$ associated with the system state x satisfies the propositional formula $L(s)$ labeling the property automaton node s.
- The initial states are $(I \times J) \cap R$.
- The transition relation includes the pairs $((s, x), (s', x')) \subseteq R \times R$, where $(s, s') \in \delta$ and $(x, x') \in \Delta$.
- The accepting states of $\mathcal{B} \times \mathcal{A}$ are $(F \times X) \cap R$. That is, a pair in R is accepting, when its $\mathcal{B}$ component is accepting.
- The labeling of any pair $(s, x) \in R$ is that of $G(x)$.

4 The Temporal Debugger

We exploit temporal specification to control stepping through different states of a concurrent system. The basic operation of a debugger is to step between different states of a system in an effective way. While doing so, one can obtain further information about the behavior of the system.

A *temporal step* consists of a finite sequence of states that satisfies some temporal property φ. Given the current global state of the system s, we are searching for a sequence $\xi = s_0 s_1 \dots s_n$ such that

- $s_0 = s$.
- n is smaller than some limit given (perhaps as a default).
- $\xi \models \varphi$.

The *temporal stack* consists of the different sequences, used in the simulation or debugging obtained so far. It contains several *temporal steps*, each corresponding to some temporal formula that was satisfied. The end state of a temporal step is also the start state of the next step. We search for a temporal step that satisfies a current temporal formula. When such a step is found, it is added to the temporal stack. We can then have several options of how to continue the search, as detailed below.

Searching a path can be done using search on pairs: a state from the joint state space of the system, and a state of the property automaton. Furthermore, each new temporal formula requires a new copy of the search space. Recursion is handled within that space. Thus, when starting the search for formula φ_1, we use one copy of the state space. When seeking a new temporal step for φ_2, we start a fresh copy. If we backtrack on the second step, we backtrack the second search, looking for a new finite sequence that satisfies φ_2. If we remove the last step, going back to the formula φ_1, we remove the second state space information, and

backtrack the first state space search. For this reason, we need to keep enough information that will enable us to resume a search after other temporal steps where exercised and backtracked.

The temporal stack contains one path, consisting of the concatenation of the various temporal steps. The last system state component of one temporal step is the first system state component of the next step. For the first step, the first system state is an initial one from the set of initial states J (we may usually assume that there is a unique initial system state). When we start the search for the kth temporal step, we translate the kth temporal property φ_k into a property automaton $\mathcal{B}_k$. We start the search for the kth temporal step with an initial state of $\mathcal{B}_k$ and the last system state on the temporal stack (the last system state of the previous state). We allow the user to observe the sequence of system states that appear on the temporal stack.

The debugging session consists of searching the system through the temporal stack. At each point we may do one of the following (see Figure 1):

- Introduce a new temporal formula and attempt to search for a temporal step from the current state. The new temporal step is added to the search stack. A new automaton for the temporal formula is created, and the product of that automaton with the system automaton with new initial state of the current state is formed. The temporal step is found by finding a path to an accepting state in this product automaton.
- Remove a step. In this case, we are back one step in the stack. We forget about the most recent temporal formula given, and can replace it by a new one in order to continue the search. We also discard the temporal automaton and product automaton generated for that temporal step.
- Backtrack the most recent step. The search process of the latest step resumes from the place it was stopped using the automaton originally created for this temporal step. This is an attempt to find another way of satisfying the last given formula. We either find a new temporal step that replaces the previous one, or report that no such step exists (in this case, we are back one step in the stack and discard the automata created for this step).
- We allow also simple debugger steps, e.g., executing one statement in one process. Such steps can be described as trivial temporal steps (using the nexttime temporal operator).

5 Stepping Modes

A debugger or a simulator allows stepping from one state to another by executing a transition enabled from the current state. Given that there are several enabled transitions, some choice is left to the user. We extend this capability and allow performing 'temporal steps', which are finite sequences of states that satisfy a given temporal formula φ. We are thus confronted with several choices:

1. The size of the step. This can be either a maximal length sequence of states (starting from the current one) that satisfies φ, or a minimal length sequence of states. We may also want to limit the possible number of states in a step, using some user-defined constant value.

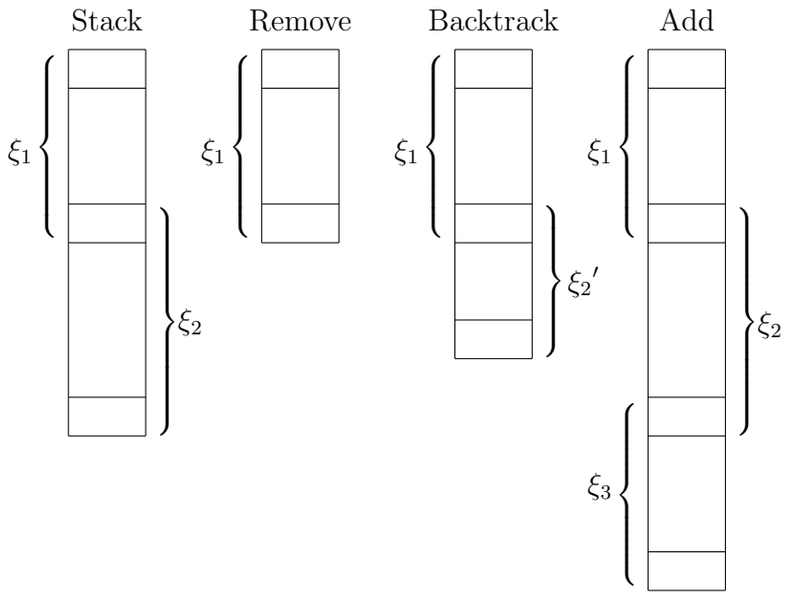

Fig. 1. Temporal stack operations

2. The transfer between different temporal steps. That is, the order in which the system finds the temporal steps. This is greatly affected by the search algorithm that is used.

The *order* between paths is the prefix order '$\sqsubseteq$'. Thus, $\rho \sqsubseteq \sigma$ if there exists ρ' such that $\sigma = \rho.\rho'$. A path generated during the search contains pairs of the form (s, x), where s is a property automaton state and x is a system state. A situation can exist, where there is an infinite sequence of increasingly bigger steps $\sigma_1 \sqsubseteq \sigma_2 \sqsubseteq \ldots$, all of them satisfying the current step formula φ. For example, consider the property $\Box p$ and a cyclic path in which all the system states satisfy p. It is possible to identify during a search when such a cycle exists and report it to the user.

Consider now a specification of type $\Box p$. A temporal step includes a sequence in which every state satisfies p. We may prefer that such a sequence will be maximal, since a longer sequence gives us more states that satisfy p. Hence more information on how p is preserved (recall that for our finite interpretation of LTL, $\Box p$ does not mean an infinite sequence in which every state satisfies p). Similarly, we may prefer that a search based on $\Diamond p$ will result in a minimal sequence that ends with a state that satisfies p. We allow the user to select between searching for a minimal or a maximal search.

Assume for the moment that the search we use is Depth First Search. Searching for a minimal temporal step starting from a pair (s, x), where s is a property automaton state, and x is a system state, is performed by by applying

DFS_min(s, x). We assume that *accept*(s) holds exactly when $s \in F$, i.e., is an accepting state of $\mathcal{B}$.

DFS_min(s, x):
if *accept*(s) then
    report sequence of system elements from stack;
    wait until Backtrack is requested;
else for each (s', x') such that $(s, s') \in \delta$, $(x, x') \in \Delta$, $G(x) \rightarrow L(s)$
    and (s', x') is new to the search, then *DFS_min*(s', x');
end *DFS_min*.

Note that if backtracking is requested by the user, i.e., an alternative temporal step, we do not attempt to continue the search from the current point. If we did, we might have found a longer path satisfying the current temporal formula, which violates the attempt to find only minimal steps.

Similarly, when searching for a maximal step, we use *DFS_max*(s, x), as follows. In this case, *saved_size* is a global variable, which maintain the size of the recursion stack from one call to the other. It is set to the current stack size when an accepting state is reached. When backtracking to an accepting state, we check whether the current stack size is the same as the one in *saved_size*. If this is the case, we did not find a longer temporal step while searching forward, and thus the current contents of the stack is a maximal step.

DFS_max(s, x):
if *accept*(s) then
    set *saved_size* to current size of recursion stack;
for each (s', x') such that $(s, s') \in \delta$, $(x, x') \in \Delta$, $G(x) \rightarrow L(s)$
    and (s', x') is new to the search, then *DFS_max*(s', x');
if *accept*(s) and *saved_size* equals current stack size then
    report sequence of system elements from stack;
    wait until Backtrack is requested;
end *DFS_max*;

Notice that we may reach a state in two directions: forward, when entering it, at the beginning of the *DFS_min* call, and backward, when backtracking from successor states. When we enter an accepting state forward, we set *saved_size* to the current size of the search stack. Upon backtracking, we check whether this variable still holds the value of the current size. If this is not the case, we must have found a longer sequence, which contains the current search stack, and satisfies the checked property. Hence the current contents of the search stack is not maximal. Note that when entering an accepting state forward, we do not check the value of *saved_size*, ignoring possible longer sequences that were generated in different search paths.

Search and Backtracking Options

There are further parameters for the choice of temporal steps, besides the minimality and maximality of the step.

- Allowing or disallowing a different step that ends with the same system state as before. In the former case, we may request an alternative step and reach exactly the same system state, but passes through a different path on the way. The latter case is easily obtained by adding a special flag to each system state that was found during the search.
- Allowing or disallowing the same sequence of system states (recall that we denoted the system as an automaton $\mathcal{A}$) to repeat. Such a repetition can happen, for example, in the following situation. The specification is of the form $(\Diamond p) \vee (\Diamond q)$. Consider a sequence of system states in which $(\neg p) \wedge (\neg q)$ holds until some state in which both p and q start to hold, simultaneously. Such a sequence can be paired up with different property automaton states to generate two different paths. Eliminating the repetition of such a sequence of system states can be obtained by keeping a tree T of nodes that participate in temporal steps reported so far (for a single given temporal step formula). Each node in the tree consists of a system state and a repetition counter (since the same state $x \in X$ can participate in one path as many times as $|S|$, the number of states of the property automaton). Each time a new temporal step is reported, the tree is updated. A new step is reported only if during the search, we deviate at least once from the paths already existing in T. Upon finding a new path, the tree T is updated.
- Allowing *all* possible paths with sequence of system states that satisfy the temporal step formula φ or only a *subset* of them. Typical searches like depth first or breadth first search do not pass through all possible paths that satisfy a given formula φ. If a state (in our case, a pair) participated before in the search, we do not continue the search in that direction. For this reason, the number of paths that can be obtained in this way is limited, and, on the other hand, the search is efficient. There are topological cases where requiring all the paths results in exponentially more paths than obtained with the above mentioned search strategies, see e.g., the case in Figure 2.

The case where similar sequences are generated as a result of repeated backtracking may seem at first to be less useful for debugging. Intuitively, we may give up exhaustiveness for the possibility of stepping through quite different sequences. However, there is a very practical case in which we may have less choice in selecting the kind of search and the effect of backtracking. Specifically, in many cases keeping several states in memory at the same time and comparing different states may be impractical. In this case, we may want to perform memoryless search, as developed for the Verisoft system [5]. In this case, we may perform breadth first search with increasingly higher depth (up to some user defined limit). We keep in the search stack only information that allows us to generate different sequences according to some order, and to regenerate a state. Such information may include the identification of the transitions that were executed from the initial states.

6 An Example

We exemplify the use of our system. Consider Dekker's solution to Dijkstra's mutual exclusion algorithm. The code for the two processes is shown in Figure 3

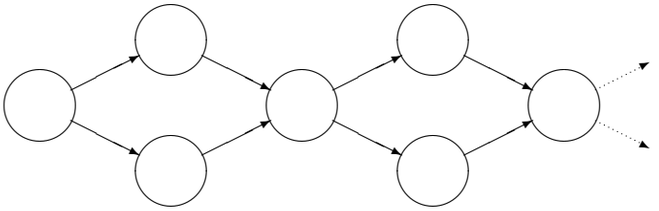

Fig. 2. Exponential number of sequences

and the corresponding flow graphs are shown in Figure 4 (the figures were generated automatically by our system, with the assistance of the DOT program [3]). We show some experiments with the temporal debugger, which allow gaining intuition about the behavior of the algorithm. Note that the critical sections of the processes $P0$ and $P1$ are labeled in Figure 4 by $m8$ and $n8$, respectively.

Process $P0$**:**

```
begin
  c0:=1;
  while true do
  begin
    c0:=0;
    while c1=0 do
    begin
        if turn=1 then
          begin
            c0:=1;
            wait turn=1;
            c0:=0
          end
    end;
    critical:=0;
    c0:=1;
    turn:=1
  end
end.
```

Process $P1$**:**

```
begin
  c1:=1;
  while true do
  begin
    c1:=0;
    while c0=0 do
    begin
        if turn=0 then
          begin
            c1:=1;
            wait turn=0;
            c1:=0
          end
    end;
    critical:=1;
    c1:=1;
    turn:=0
  end
end.
```

Fig. 3. Code for Dekker's Algorithm

At first look at the code, we may be able to identify that the variable *turn* is assigned according to the process that has priority to get to the critical section. The variable *ci* for $i = 0, 1$ is set to 1 when a process does not attempt to get to the critical section (or does not insist on doing that) and 0 otherwise. The system provides us with automatic translation of the code to a flow graph, and

a labeling of the nodes. We use the notation Pi_at_l to denote the predicate that asserts that process Pi is at label l.

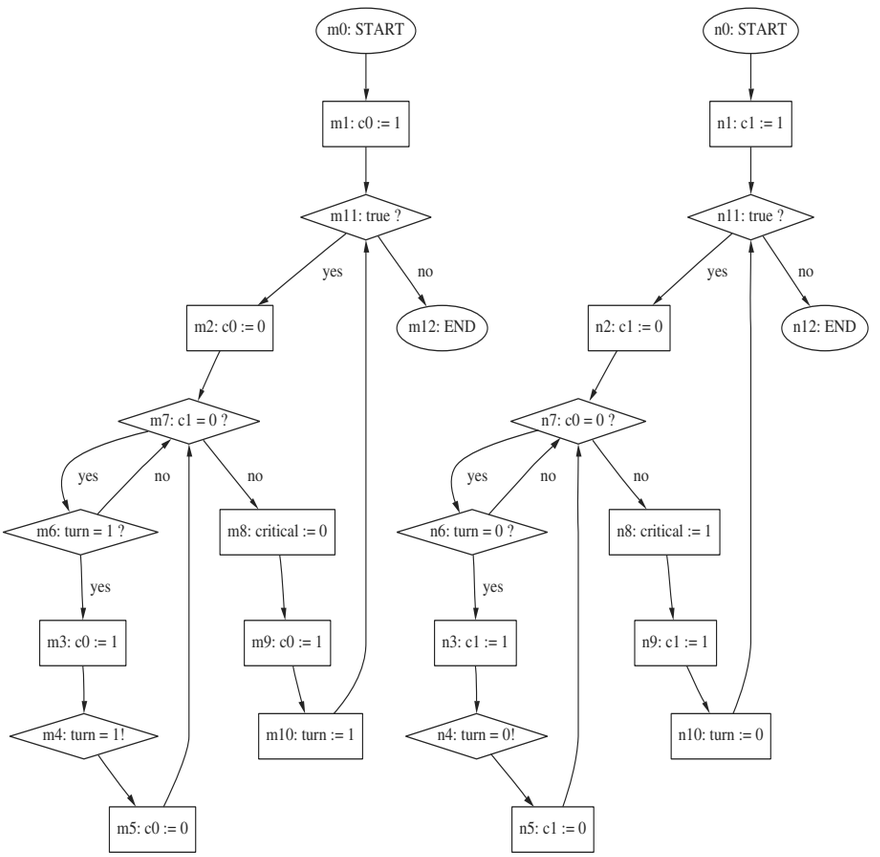

Fig. 4. Dekker's mutual exclusion solution

We can start the debugging with an attempt to understand what happens when process $P0$ attempts to enter and $P1$ does not. We use the formula $\varphi_1 = \Diamond(P0\_at\_m2 \wedge P1\_at\_n1)$ and search for a shortest temporal step satisfying this. (Our interpretation of *at*, i.e., that we are at a node, is that the node was the last to be executed in that process.) We check that at this point, $c0 = 0 \wedge c1 = 1$. We can continue from here by using the formula $\varphi_2 = \Box P1\_at\_n1$, i.e., asserting that process $P1$ does not move. We can use a search that will step through all the states that end a finite sequence satisfying this formula. Alternatively, we may remove the last temporal step and choose $\varphi_2 = (\Box P1\_at\_n1) \wedge (\Diamond P0\_at\_m8)$

to check whether we can get to the critical section without progressing the $P1$ process.

Consider the case where both processes want to get into the critical section. Initially, we have $turn = 0$ (the global initialization is being performed using a separate process). We clean the temporal stack, and use $\varphi_1 = \Diamond(P0\_at\_m2 \wedge P1\_at\_n2)$. Again, we can check if we can get to the critical section without process $P1$ moving, using the formula $\varphi_2 = (\Box P1\_at\_n2) \wedge (\Diamond P0\_at\_m8)$. This does not succeed in providing any sequence. We can remove the last temporal step and use the formula $\varphi_2 = \Box P1\_at\_n2$. As a result, we obtain only the states $m6$ and $m7$. We now remove the last temporal step. We would like to check whether we can get to the critical section of $P0$. We can check that by using $\varphi_2 = \Diamond P0\_at\_m8$. Observing the paths obtained, we may gain information about the way we can gain access to the critical section, namely by process $P1$ progressing to $n3$, relinquishing its attempt to gain access, by setting $c1 = 1$. Then $P0$ may exit the loop on $m6$ and $m7$, and enter the critical section at $m8$. We may also want to break this path into several smaller temporal steps, in order to understand this access better.

We can check other possibilities. For example, after getting to the first state in the execution where $P0\_at\_m2$ and $P1\_at\_n2$, while $turn = 0$, we can check which process can get to the critical section first. Checking $\neg P0\_at\_m8 \,\mathcal{U}\, P1\_at\_n8$ will not result in any path, while checking $\neg P1\_at\_n8 \,\mathcal{U}\, P0\_at\_m8$ will show a path in which $P0$ gets first to its critical section. Similarly, we can check whether one process can get to its critical section again before the other process was able to do so from various states in the execution.

7 Implementation

The implementation of the program for our debugger is written in two top-level pieces: a graphical user interface written in Tcl/Tk, and a back end computing engine written in SML. The language SML is a higher-order applicative language with restricted imperative features.

There are several major components to the back end computing engine. We must be able to translate each of the given LTL formula and the parallel object programs into finite state automata, and then compute their restricted product. Having created the restricted product automaton, we must be able to perform a variety of different searches on it.

Because we are likely to need to construct a number of finite state automata in the course of a single debugging session, and each of these automata is likely to be quite large, we take advantage of the higher-order applicative nature of SML to build a generic lazy implementation of these automata. The implementation is parameterized by a type of state information, an initial state, a function for determining when two states should be treated as the same, and a function which, for a given state, computes a list of the states pointed to from the current state.

Initially, to create the desired automata, we create an initial node containing the initial state information and a continuation function which will create the out edges to the next states including a continuation function for each of them to create their out edges. Every time we visit a state node, if its out edges have

not yet been constructed, then we apply the continuation to create the adjacent states and we update the node with the new information about its out edges. These details are encapsulated in a abstraction hiding the details of the lazy structure from subsequent search algorithms.

As a result of the lazy nature of the construction of the nodes in the various automata we need, as we explore further and further along possible execution paths in the concurrent system, increasingly more of the state space for the system is constructed, potentially growing until the full automaton has been realized. However, for each of the LTL formulae that we use in taking temporal steps, we only construct as much of the automaton corresponding to the LTL formula and as much of the restricted product automaton as is necessary to find the path, making up the desired temporal step. In the worst case, we could be forced to unfold the full automata, but in general there should be considerable space saving achieved by not expanding all the nodes.

8 Discussion

Temporal logic in conjunction with a search is employed by *model checking* [1,2] techniques. There, we want to check whether all executions (sometimes including infinite ones) starting with a given system state (usually an initial state) satisfy a given property. In our context, we are using temporal specification is a different way, to control the stepping between system states. We are looking for finite sequences of states that satisfy a given temporal specification, and move the current control to the last state of the sequence.

In some sense, our approach is related to the *choppy* temporal logic of Pnueli and Rosner [8]. There, one can use temporal specification over finite sequences and combine them using the *chop* ($\mathcal{C}$) operator. We are effectively stepping through different finite sequences and progressing through the execution. Note that in the temporal semantics of [8], $\varphi_1 \mathcal{C} \varphi_2$ holds for a path that concatenates two shorter paths, where the first satisfies φ_1 and the second satisfies φ_2, respectively. In our case, the last state of one temporal step is the first state of the next step. Thus, to obtain the same effect as in the choppy logic, we may want to use φ_1 and $\bigcirc\varphi_2$.

We could have bundled different temporal steps into an equivalent linear temporal property about the entire system. Then we could perform LTL model checking as in SPIN [6]. The property would not look the same in the standard version of temporal logic, since there is no operator that sequentially combines finite temporal assertions. In this case, we either obtain a confirmation for the property, or a single (often lengthy) counterexample that starts from the initial state. In our approach, we examine the behavior of the system in a stepwise manner, and, through the developed tool, were capable of keeping track of the current state, allowing us to zoom quickly into potential programming errors. In fact, we suggest our approach as an extension to LTL based model checkers such as SPIN.

References

1. E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic. Workshop on Logic of Programs, Yorktown Heights, NY, Lecture Notes in Computer Science 131, Springer-Verlag, 1981, 52–71.
2. E. A. Emerson, E. M. Clarke, Characterizing correctness properties of parallel programs using fixpoints, International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 85, Springer-Verlag, July 1980, 169–181.
3. E.R. Gansner, S.C. North, An open graph visualization system and its applications to software engineering, Software – Practice and Experience, 30(2000), 1203–1233.
4. R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, *PSTV95, Protocol Specification Testing and Verification*, 3–18, Chapman & Hall, 1995, Warsaw, Poland.
5. P. Godefroid, Model checking for programming languages using Verisoft, POPL 1997, 174–186.
6. G. Holzmann, Design and Validation of Computer Protocol, *Prentice Hall*.
7. A. Pnueli, The temporal logic of programs, 18th IEEE symposium on Foundation of Computer Science, 1977, 46–57.
8. A. Pnueli, R. Rosner, A Choppy Logic, Logic in Computer Science 1986, Cambridge, Massachusetts, 1986, 306–318.