# Temporal Debugging for Concurrent Systems

Elsa Gunter
Department of Computer Sceince
New Jersey Institute of Technology

Doron Peled
Dept. of Elect. and Computer Eng.
University of Texas at Austin

October 15, 2001

**Abstract**

## 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 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 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 [3] 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.

## 2 Defining LTL on Finite Sequences

One of the most popular specification formalisms for concurrent and reactive systems is Linear Temporal Logic (LTL) [4]. Its syntax is as follows:

$\varphi ::= (\varphi_1) \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\, \varphi_1 \mid \overline{\bigcirc}\varphi_1 \mid \Box\varphi_1 \mid \Diamond\varphi_1 \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \varphi_1 \,\mathcal{V}\, \varphi_2 p$

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

- $\sigma \models \bigcirc\varphi$ iff $|\sigma| > 1$ and $\sigma^{(1)} \models \varphi$.
- $\sigma \models \Box\varphi$ iff for each $0 \leq i < |\sigma|$, $\sigma^{(i)} \models \varphi$.
- $\sigma \models \Diamond\varphi$ iff there exists $0 \leq i < |\sigma|$ such that $\sigma^{(i)} \models \varphi$.
- $\sigma \models \varphi \,\mathcal{U}\, \psi$ 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 $\sigma \models \psi$.

The rest of the operators can be defined using the above operators. In particular, $\overline{\bigcirc}\varphi = \neg \bigcirc \neg\varphi$, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi \,\mathcal{V}\, \psi = \neg((\neg\varphi) \,\mathcal{U}\, (\neg\psi))$. The operator $\overline{\bigcirc}$ is a 'weak' version of the $\bigcirc$ operator. Whereas $\bigcirc\varphi$ means that $\varphi$ 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 $\varphi$.

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 $\overline{\bigcirc}$*false* holds in a state that is in deadlock termination.

The operators $\mathcal{U}$ and $\mathcal{V}$ can be characterized using a recursive equation, which is useful for understanding the transformation algorithm, presenting 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 [3] for translating an LTL formula $\varphi$ 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 $L : S \rightarrow D$ 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 $\varphi$ 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*. Next, we 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 $\Box\psi = \mathbf{False}\,\mathcal{V}\,\psi$.

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 $\varphi$ 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 $\eta$ 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*, *incoming* and *strong* retain their previous values in both copies. The algorithm continues recursively with the split 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. 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 | new left | next left | set strong left | new right | next right |
|---|---|---|---|---|---|
| $\mu\,\mathcal{U}\,\eta$ | $\{\mu\}$ | $\{\mu\,\mathcal{U}\,\eta\}$ | $\surd$ | $\{\eta\}$ | $\emptyset$ |
| $\mu\,\mathcal{V}\,\eta$ | $\{\eta\}$ | $\{\mu\,\mathcal{V}\,\eta\}$ | | $\{\mu, \eta\}$ | $\emptyset$ |
| $\mu \vee \eta$ | $\{\mu\}$ | $\emptyset$ | | $\{\eta\}$ | $\emptyset$ |
| $\mu \wedge \eta$ | $\{\mu, \eta\}$ | $\emptyset$ | | — | — |
| $\overline{\bigcirc}\mu$ | $\emptyset$ | $\mu$ | | — | — |
| $\bigcirc\mu$ | $\emptyset$ | $\mu$ | $\surd$ | — | — |

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 $\varphi$ as follows. The states $S$ are the nodes in *completed-nodes*. Let $P$ be the set of propositions that appear in the formula $\varphi$. 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 $\delta$ includes pairs of nodes $(s, s')$ if $s$ belongs to the field *init* of $s'$. The accepting (final) states satisfy the following:

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

The accepting condition for infinite sequences is an extended form of *generalized Büchi automata*. As in generalized Büchi automata, an infinite sequence is accepting if it traverses at least one node per accepting set infinitely often. A finite sequence is accepting if it ends in a state from the finite accepting set.

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 $\varphi$).

We also denote the system automaton as $\mathcal{A} = \langle X, J, \Delta, D, G \rangle$, where $X$ are the states, $J \subseteq X$ are the initial states, $\Delta \subseteq X \times X$ is the transition relation, $D$ are the set of labels (same as for $\mathcal{B}$), and $G : X \to D$ is the labeling function. Here there are no accepting states. The automata product $\mathcal{B} \times \mathcal{A}$ has the following components:

- The states $R$ are a subset of $S \times X$ where $L(s) = G(X)$, i.e., $\{(s,x)|L(s) = G(x)\}$.
- The initial states are $(I \times J) \cap R$.
- The transition relation includes the pairs $((s,x),(s',x'))$, where $(s,s') \in \delta$, $(x,x') \in \Delta$.
- The accepting states of $\mathcal{B} \times \mathcal{A}$ are $(S \times F) \cap R$. 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 the same as $L(s)$ (which is the same as $G(x)$.

## 4 The Temporal Debugger

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

A *temporal step* consists of a finite sequence of states that satifies some terporal propoerty $\varphi$. Given the current global state of the system $s$, we are searching for a sequence $\xi = s_0 s_1 \ldots s_n$ such that

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

The *termporal stack* consists of the different sequences, used in the simulation obtained so far. It contains several *temporal steps*, each corresponding to some temporal formula that was satified. The end state of a temporal step is also the start state of the next step.

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

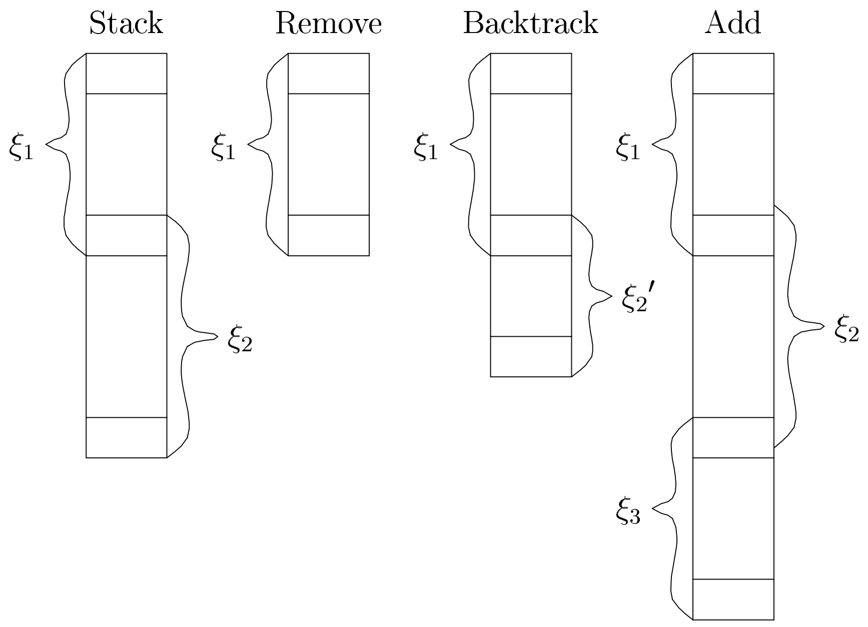

Figure 1: Temporal stack operations

- Introduce a new termpoal formula and attempt to search for a temporal state from the current state. The new temporal is added to the search stack.
- Romove 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.
- Backtrack the most recent step. The depth first search process of the latest step resumes from the place it was stopped. 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).
- We allow also simple debugger steps, e.g., executing one statemet in one process. Such steps can be described as trivial temporal steps (using the nexttime temporal operator).

Note the following: Searching a path can be done using depth first seach on pairs: a state from the joint state space of the programs, and a state of the property automaton. Furthermore, each new temporal formula requires a new depth first search space. Recursion is handled within that space. Thus, when starting the depth first search for formula $\varphi_1$, we use one copy of the state space. When seeking a new temporal step for $\varphi_2$, we start a fresh copy. If we backtrack the second step, we backtrack the second depth first search, looking for a new fiinite sequence that satisfies $\varphi_2$. If we remove the last step, going back to the formula $\varphi_1$, we remove the second state space information, and backtrack the first state space. On the other hand, the temporal stack contains one path, consisting of the concatenation of the various temporal steps. We may want to display the path obtained by the search.

## 5 Stepping Modes

A debugger or a simulator allows to step 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 to perform 'temporal steps', which are finite sequences of states that satisfy a given temporal formula $\varphi$. 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 $\varphi$, or
   - a minimal length sequence of states.
2. The move between different temporal steps. That is, the order in which the system presents the temporal steps. This is greatly affected by the search algorithm that is used.

We need to define precisely what does a minimal and a maximal path mean. The *order* between paths is the prefix order '$\sqsubseteq$'. Thus, $\rho \sqsubseteq \sigma$ if there exists $\rho'$ 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 satisfy the current step formula $\varphi$. For example, consider the property $\Box p$ and a cyclic path in which all the system states satisfy $p$. Since a step has to be finite, we will restrict the search to a maximal path that does not contain the same pair twice.

### Polarity of a Temporal Step

To illustrate the first point, consider a specification of type $\Box p$. A temporal step includes a sequence in which every state satisfies $p$. Intuitively, we would like such a sequence to be maximal, since a longer sequence gives us more states and hence more information on how $p$ is preserved (reacall that for finite LTL, $\Box p$ does not mean an infinite sequence in which every state satisfies $p$. Similarly, $\Diamond p$ should 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.

When we find a temporal step that satisfies the current temporal formula, i.e., when the current property automaton $\mathcal{B}$ state $s$ is accepting (denoted by *accept*$(s)$), we report the sequence of system states that are in the search stack.

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 perfomed 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 foreach $(s', x')$ such that $(s, s') \in \delta$, $(x, x') \in \Delta$
    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. 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.

```
DFS_max(s, x):
if accept(s) then
    set $saved - size$ to current size of recursion stack;
foreach $(s', x')$ such that $(s, s') \in \delta$, $(x, x') \in \Delta$
    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;
```

### 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 step as before. In the former case, we may request an alternative step and reach exactly the same system state, but pass 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 to repeat. Such a repetition can happen 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 partitipate 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 a path as many times as $|S|$. 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 in $T$.

- Allowing all possible paths with sequence of system states that satisfy the temporal step formula $\varphi$ 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 $\varphi$. 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.

## 6 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 program state (usually an intial state) satisfy a given property. In our context, we are using temporal specification is a different way, to control the stepping between program states. We are looking

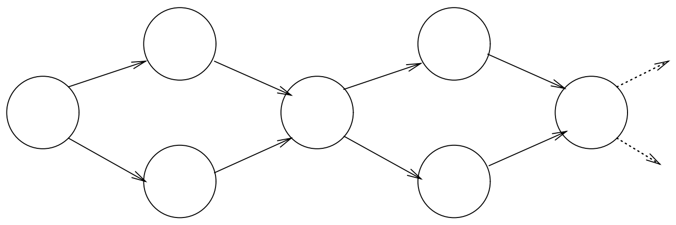

Figure 2: Exponential number of sequences

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 [5]. 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 [5], $\varphi_1 \mathcal{C} \varphi_2$ holds for a path that concatenates two shorter paths, where the first satisfies $\varphi_1$ and the second satisfies $\varphi_2$, 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 $\varphi_1$ and $\bigcirc\varphi_2$.

## 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] 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.

[4] A. Pnueli, The temporal logic of programs, 18th IEEE symposium on Foundation of Computer Science, 1977, 46–57.

[5] A. Pnueli, R. Rosner, A Choppy Logic, Logic in Computer Science 1986, Cambridge, Massachusetts, 1986, 306–318.