

**AUTOMATIC VERIFICATION OF FINITE
STATE CONCURRENT SYSTEMS USING
TEMPORAL LOGIC SPECIFICATIONS**

E. M. Clarke, E. A. Emerson and A. P. Sistla

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-31 November 1985

A preliminary version of this paper was presented at the *1983 ACM Symp. on Principles of Programming Languages*. This slightly revised version is to appear in *ACM Transactions on Programming Languages*.

Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications

E. M. Clarke
Carnegie Mellon University, Pittsburgh

E. A. Emerson
University of Texas, Austin

A. P. Sistla
GTE Laboratories, Inc., Waltham

Categories and Subject Descriptors: D.1.3 [Concurrent Programming], D.2.4 [Program Verification], F.3.1 [Specifying, Verifying, and Reasoning about Programs], F.4.1 [Mathematical Logic].

General Terms: Program Verification, Concurrent Systems, Temporal Logic, Network Protocol Verification.

Additional Key Words and Phrases: Temporal Logic Model Checking, Computation Tree Logic, Finite State Concurrent Systems.

Abstract: We give an efficient procedure for verifying that a finite state concurrent system meets a specification expressed in a (propositional, branching-time) temporal logic. Our algorithm has complexity linear in both the size of the specification and the size of the global state graph for the concurrent system. We also show how this approach can be adapted to handle fairness. We argue that our technique can provide a practical alternative to manual proof construction or use of a mechanical theorem prover for verifying many finite state concurrent systems. Experimental results show that state machines with several hundred states can be checked in matter of seconds.

The first and third authors were supported by NSF Grant MCS-815553. The second author was partially supported by a University of Texas Summer Research Award, a departmental grant from IBM, and NSF Grant MCS-8302878.

1. Introduction.

In the traditional approach to concurrent program verification, the proof that a program meets its specification is constructed by hand using various axioms and inference rules in a deductive system such as temporal logic ([13], [9], [15]). The task of proof construction is in general quite tedious, and a good deal of ingenuity may be required to organize the proof in a manageable fashion. Mechanical theorem provers have failed to be of much help due to the inherent complexity of testing validity for even the simplest logics.

We argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic. The global state graph of the concurrent system can be viewed as a finite Kripke structure, and an efficient algorithm can be given to determine whether a structure is a model of a particular formula - i.e. to determine if the program meets its specification. The algorithm, which we call a *model checker*, is similar to the global flow analysis algorithms used in compiler optimization and has complexity linear in both the size of the structure and the size of the specification. When the number of global states is not excessive (i.e. not more than a few thousand) we believe that our technique may provide a useful new approach to the verification of finite state concurrent systems.

Our approach is of wide applicability since a large class of concurrent programming problems have finite state solutions, and the interesting properties of many such problems can be specified in propositional temporal logic. For example, many network communication protocols (e.g. the Alternating Bit Protocol [2]) can be modeled at some level of abstraction by a finite state system. A typical requirement for such systems is that every transmitted message must ultimately be received; this can easily be expressed in the logic we use.

Our specification language is a propositional, branching-time temporal logic called *Computation Tree Logic* (CTL) and is similar to the logical systems described in [4], [1], and [3]. Since our goal is to specify concurrent systems we must be able to assert that a correctness property only holds on fair execution sequences. It follows from the results of ([4], [5]) that CTL cannot express such a property. The alternative of using a linear time logic is ruled out because any model checker for such a logic must have high complexity ([18]). We overcome this problem by moving fairness requirements into the semantics of CTL. Specifically, we change the definition of our basic modalities so that only fair paths are considered. Our previous model checking algorithm is modified to handle this extended logic without changing its complexity.

Our paper is organized as follows: Section 2 contains the syntax and semantics of our logic. In section 3 we describe the basic model checking algorithm and illustrate its use to establish absence of starvation for a solution to the mutual exclusion problem. An extension of the model checking algorithm which only considers *fair computations* is given in section 4. Section 5 describes an experimental implementation of the

extended model checking algorithm and shows how it can be used to verify the correctness of the Alternating Bit Protocol. In section 6 we consider extensions of our logic that are more expressive and investigate the complexity of model checkers for these logics. The paper concludes with a discussion of related work and remaining open problems.

2. The Specification Language.

The formal syntax for CTL is given below. AP is the underlying set of *atomic propositions*.

1. Every atomic proposition $p \in AP$ is a CTL formula.
2. If f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, AXf_1 , EXf_1 , $A[f_1 U f_2]$, and $E[f_1 U f_2]$.

The symbols \wedge and \neg have their usual meanings. X is the *nexttime* operator; the formula AXf_1 (EXf_1) intuitively means that f_1 holds in every (in some) immediate successor of the current program state. U is the *until* operator; the formula $A[f_1 U f_2]$ ($E[f_1 U f_2]$) intuitively means that for every computation path (for some computation path), there exists an initial prefix of the path such that f_2 holds at the last state of the prefix and f_1 holds at all other states along the prefix.

We define the semantics of CTL formulas with respect to a labeled state-transition graph. Formally, a CTL structure is a triple $M = (S, R, P)$ where

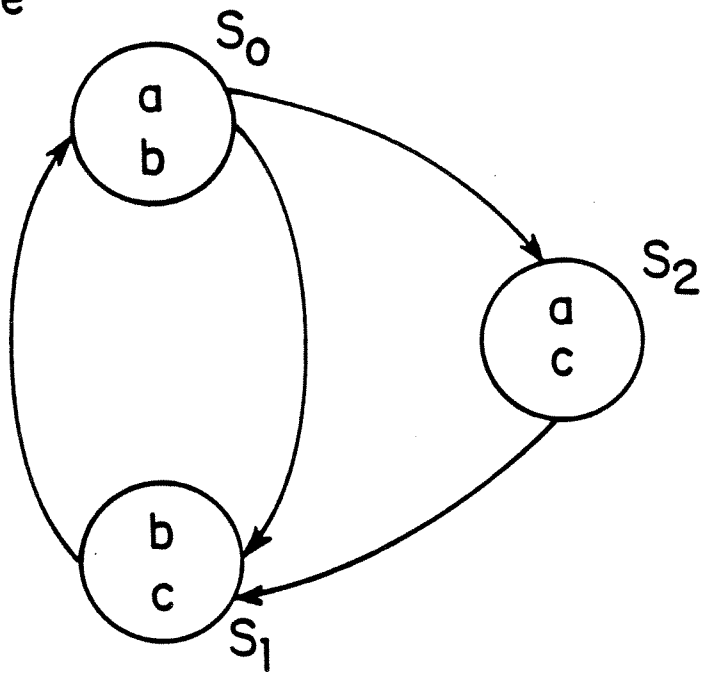
1. S is a finite set of states.
2. R is a binary relation on S ($R \subseteq S \times S$) which gives the possible transitions between states and must be total, i.e. $\forall x \in S \exists y \in S [(x,y) \in R]$.
3. $P : S \rightarrow 2^{AP}$ assigns to each state the set of atomic propositions true in that state.

A *path* is an infinite sequence of states (s_0, s_1, s_2, \dots) such that $\forall i [(s_i, s_{i+1}) \in R]$. For any structure $M = (S, R, P)$ and state $s_0 \in S$, there is an *infinite computation tree* with root labeled s_0 such that $s \rightarrow t$ is an arc in the tree iff $(s, t) \in R$. Figure 2.1 shows a CTL structure and the associated computation tree rooted at s_0 .

We use the standard notation to indicate truth in a structure: $M, s_0 \models f$ means that formula f holds at state s_0 in structure M . When the structure M is understood, we simply write $s_0 \models f$. The relation \models is defined inductively as follows:

$$\begin{aligned} s_0 \models p & \quad \text{iff } p \in P(s_0). \\ s_0 \models \neg f & \quad \text{iff } \text{not}(s_0 \models f). \\ s_0 \models f_1 \wedge f_2 & \quad \text{iff } s_0 \models f_1 \text{ and } s_0 \models f_2. \end{aligned}$$

A structure



The corresponding tree
for start state S_0

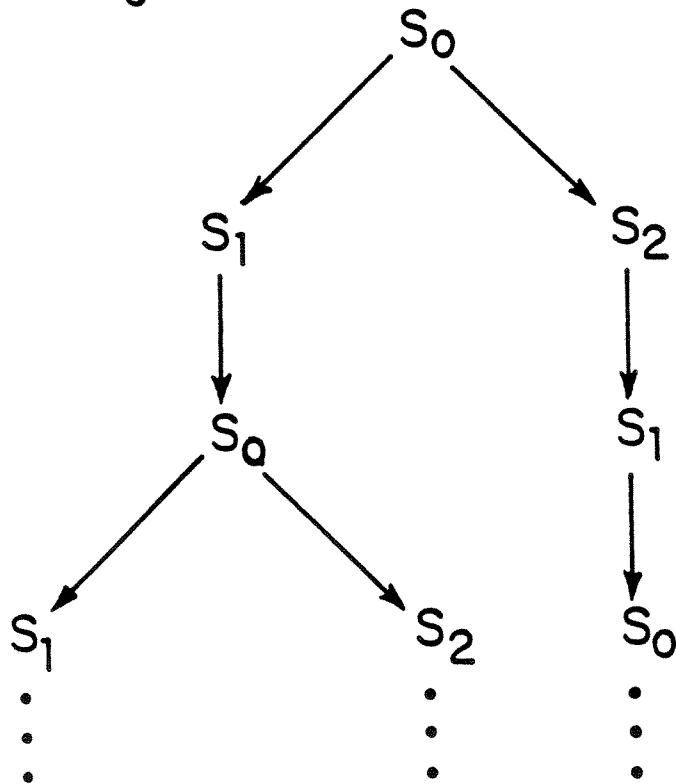


Figure 2.1

$s_0 \models \text{AX}f_1$ iff for all states t such that $(s_0, t) \in R$, $t \models f_1$.

$s_0 \models \text{EX}f_1$ iff for some state t such that $(s_0, t) \in R$, $t \models f_1$.

$s_0 \models \text{A}[f_1 \text{ U } f_2]$ iff for all paths (s_0, s_1, \dots) , $\exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \rightarrow s_j \models f_1]]$.

$s_0 \models \text{E}[f_1 \text{ U } f_2]$ iff for some path (s_0, s_1, \dots) , $\exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \rightarrow s_j \models f_1]]$.

We will also use the following abbreviations in writing CTL formulas:

$\text{AF}(f) \equiv \text{A}[\text{True U } f]$ intuitively means that f holds in the future along every path from s_0 , i.e., f is *inevitable*.

$\text{EF}(f) \equiv \text{E}[\text{True U } f]$ means that there is some path from s_0 that leads to a state at which f holds, *potentially* holds.

$\text{EG}(f) \equiv \neg \text{AF}(\neg f)$ means that there is some path from s_0 on which f holds at every state.

$\text{AG}(f) \equiv \neg \text{EF}(\neg f)$ means that f holds at every state on every path from s_0 , i.e., f holds *globally*.

Figure 2.2 shows how some simple correctness properties would be represented using these operators.

The global state transition graphs of many concurrent programs can be modeled as CTL structures. For example Figure 2.3 shows the CTL structure for a simple solution to the *mutual exclusion problem* for two processes P_1 and P_2 . In this solution each process is always in one of three regions of code:

N_i the Noncritical region,

T_i the Trying region,

or C_i the Critical region.

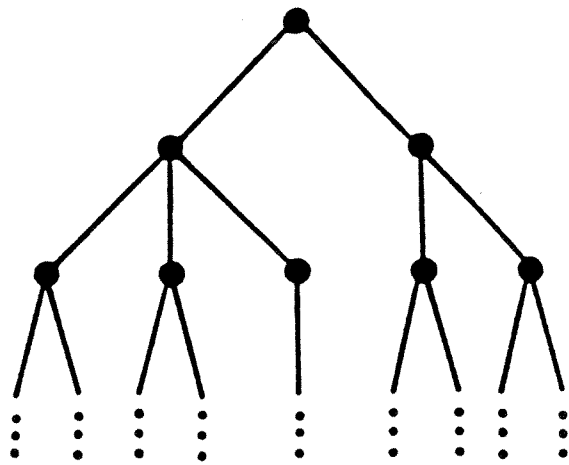
Note that we only record transitions between different regions of code; moves entirely within the same region are not considered at this level of abstraction. Also each transition is due to the execution of a step of exactly one process. It is easy to see, in this case, that $\text{AF}(C_1)$ is true in state one and that $\text{EF}(C_1 \ \& \ C_2)$ is false in state zero.

3. Model Checker

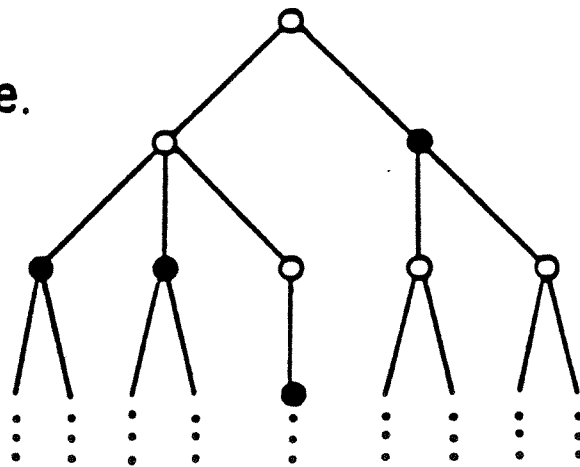
Assume that we wish to determine whether formula f_0 is true in the finite structure $M = (S, R, P)$. We design our algorithm to operate in stages: The first stage processes all subformulas of f_0 of length 1, the second stage processes all subformulas of length 2, etc. At the end of the i^{th} stage, each state will be labeled with the set of all subformulas of length less than or equal to i that are true in the state. We let the expression

AGp : p is invariant.

● = p
○ = ¬p



AFp : p is inevitable.



EFp : p potentially holds.

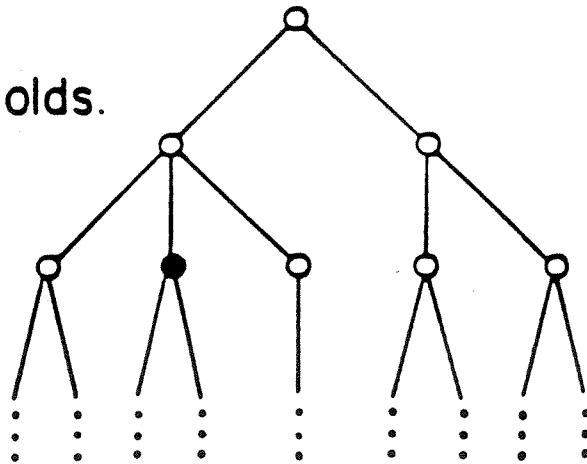


Figure 2.2

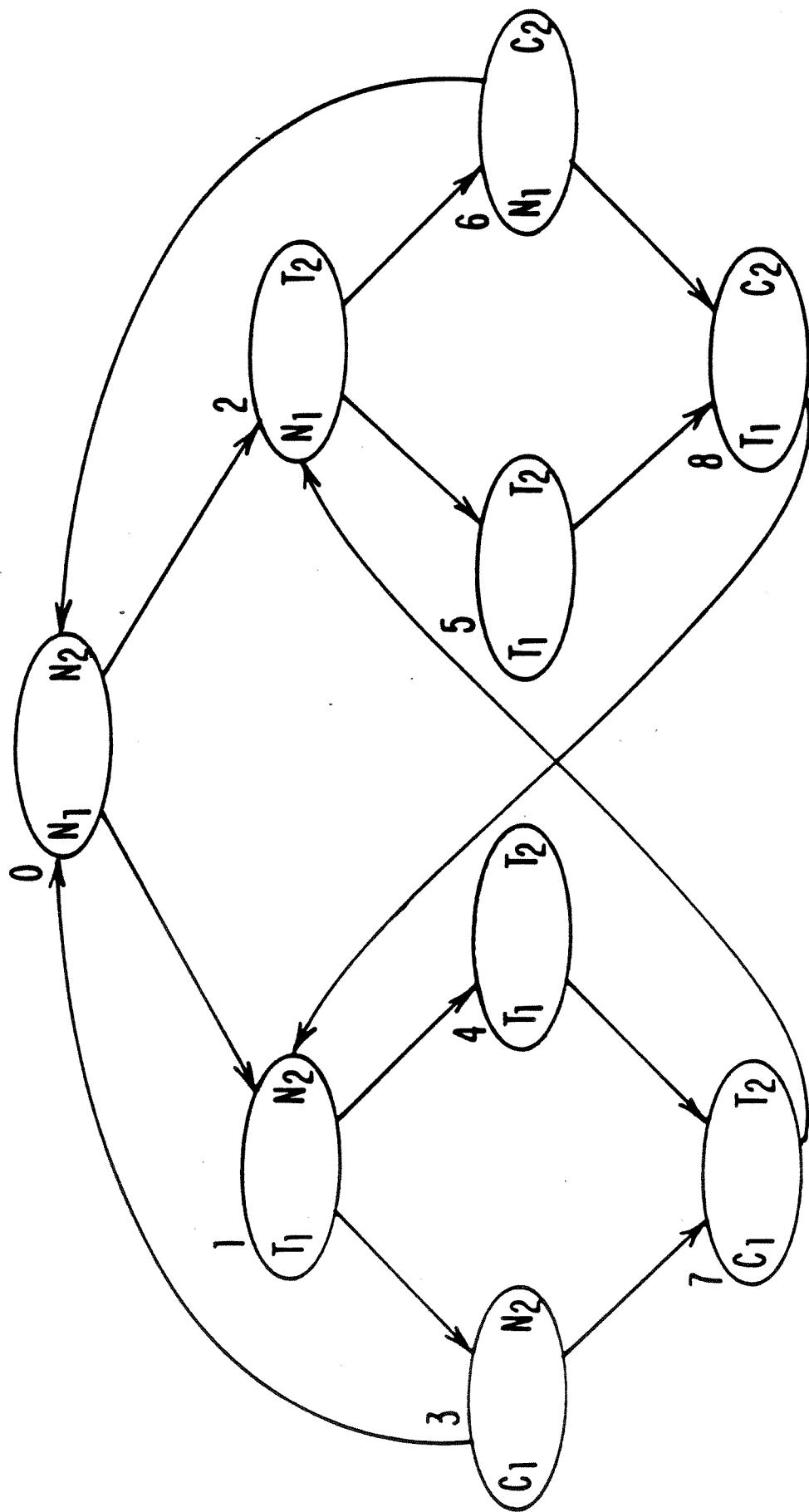


Fig. 2.3 : Global state transition graph for two process mutual exclusion problem.

label(s) denote this set for state s. When the algorithm terminates at the end of stage $n = \text{length}(f_0)$, we see that for all states s, $M, s \models f$ iff $f \in \text{label}(s)$ for all subformulas f of f_0 .

We will use the following primitives for manipulating formulas and accessing the labels associated with states:

- $\text{arg1}(f)$ and $\text{arg2}(f)$ give the first and second arguments of a two argument temporal operator; thus, if f is $A[f_1 \cup f_2]$, then $\text{arg1}(f) = f_1$ and $\text{arg2}(f) = f_2$.
- $\text{labeled}(s, f)$ will return true (false) if state s is (is not) labeled with formula f.
- $\text{add_label}(s, f)$ adds formula f to the current label of state s.

Our state labelling algorithm (**procedure** $\text{label_graph}(f)$) must be able to handle seven cases depending on whether f is atomic or has one of the following forms: $\neg f_1$, $f_1 \wedge f_2$, AXf_1 , EXf_1 , $A[f_1 \cup f_2]$, or $E[f_1 \cup f_2]$. We will only consider the case in which $f = A[f_1 \cup f_2]$ here since all of the other cases are either straightforward or similar. For the case $f = A[f_1 \cup f_2]$ our algorithm uses a *depth first search* to explore the state graph. The bit array $\text{marked}[1: nstates]$ is used to indicate which states have been visited by the search algorithm. ST is an auxiliary stack variable introduced for the proof of correctness of the algorithm. The boolean procedure $\text{stacked}(s)$ indicates whether state s is currently on the stack ST.

```

procedure  $\text{label\_graph}(f)$ 
begin
    ...
    {main operator is AU}
    begin
        ST := empty_stack;
        for all  $s \in S$  do  $\text{marked}(s) := \text{false}$ ;
        L : for all  $s \in S$  do
            if  $\neg \text{marked}(s)$  then  $\text{au}(f,s,b)$ 
        end
    end
    ...
end

```

The recursive procedure $\text{au}(f,s,b)$ performs the search for formula f starting from state s. When au terminates, the boolean result parameter b will be set to true iff $s \models f$. The annotated code for procedure au is shown below:

```

procedure  $\text{au}(f,s,b)$ 
begin

```

```

    {Assume that s is marked. If s is already labeled with f we set b to true and return. Otherwise, if s is
    on the stack, then we have found a cycle in the state graph on which  $\text{arg1}(f)$  holds but f is never
    fulfilled (see lemma 3.2 in the Appendix I). Thus, we set b to false and return. Otherwise, we have

```

already completed a depth first search from s , and f is false at s ; so we must also set b to false and return in this case. Note that there is no need to distinguish between the last two cases, since the action is the same in each case.}

```

if marked(s) then
  begin
    if labeled(s,f) then
      begin b := true; return end;
    b := false; return
  end;

```

{Mark state s as visited. Let $f = A[f_1 \cup f_2]$. If f_2 is true at s , f is true at s ; so label s with f and return true. If f_1 is not true at s , then f is not true at s ; so return false. }

```

marked(s) := true;
if labeled(s,arg2(f)) then
  begin add_label(s,f); b := true; return end
else if ¬labeled(s,arg1(f)) then
  begin b := false; return end;

```

{Now we know that f_1 is true at s and that f_2 is not. Check to see if f is true at all successor states of s . If there is some successor state s_1 at which f is false, then f is false at s also; hence remove s from the stack and return false. If f is true for all successor states, then f is true at s ; so remove s from the stack, label s with f , and return true. (We remind the reader that ST is an auxiliary variable which is used in the correctness proof given in Appendix I.)}

```

push(s,ST);
for all  $s_1 \in$  successors(s) do
  begin
    au (f, $s_1$ , $b_1$ );
    if ¬ $b_1$  then
      begin pop(ST); b := false; return end
  end;
pop(ST); add_label(s,f); b := true; return

```

end of procedure au.

A formal proof of the correctness of this part of the algorithm is given in Appendix I. Assuming that the states of the graph are already correctly labeled with f_1 and f_2 , it is easy to see that the above algorithm requires time $O(\text{card}(S) + \text{card}(R))$. The time spent by one call of procedure au excluding the time spent in recursive calls is a constant plus time proportional to the number of edges leaving the state s . Thus, all calls to au together require time proportional to the number of states plus the number of vertices since au is called at most once in any state.

To handle formulas of the form $f = E[f_1 \cup f_2]$ we first find all of those states which are labeled with f_2 . We then work backwards using the converse of the successor relation and find all of the states that can be reached

by a path in which each state is labeled with f_1 . All such states should be labeled with f . Formal proof of this case will be left to the reader.

We next show how to handle CTL formulas with arbitrary nesting of subformulas. Note that if we write formula f in prefix notation and count repetitions, then the number of subformulas of f is equal to the length of f . (The length of f is determined by counting the total number of operands and operators.) We can use this fact to number the subformulas of f . Assume that formula f is assigned the integer i . If f is unary i.e. $f = (\text{op } f_1)$ then we assign the integers $i+1$ through $i + \text{length}(f_1)$ to the subformulas of f_1 . If f is binary i.e. $f = (\text{op } f_1 f_2)$ then we assign the integers from $i + 1$ through $i + \text{length}(f_1)$ to the subformulas of f_1 and $i + \text{length}(f_1)$ through $i + \text{length}(f_1) + \text{length}(f_2)$ to the subformulas of f_2 . Thus, in one pass through f we can build two arrays $\text{nf}[1 : \text{length}(f)]$ and $\text{sf}[1 : \text{length}(f)]$ where $\text{nf}[i]$ is the i^{th} subformula of f in the above numbering and $\text{sf}[i]$ is the list of the numbers assigned to the immediate subformulas of the i^{th} formula. For example, if $f = (\text{AU } (\text{NOT } X) (\text{OR } Y Z))$, then nf and sf are given below:

| | | | |
|----------------|---|----------------|--------------|
| $\text{nf}[1]$ | $(\text{AU } (\text{NOT } X) (\text{OR } Y Z))$ | $\text{sf}[1]$ | $(2\ 4)$ |
| $\text{nf}[2]$ | $(\text{NOT } X)$ | $\text{sf}[2]$ | (3) |
| $\text{nf}[3]$ | X | $\text{sf}[3]$ | nil |
| $\text{nf}[4]$ | $(\text{OR } Y Z)$ | $\text{sf}[4]$ | $(5\ 6)$ |
| $\text{nf}[5]$ | Y | $\text{sf}[5]$ | nil |
| $\text{nf}[6]$ | Z | $\text{sf}[6]$ | nil |

Given the number of a formula f we can determine in constant time the operator of f and the number assigned to its arguments. We can also efficiently implement the procedures "labeled" and "add_label". We associate with each state s a bit array $L[s]$ of size $\text{length}(f)$. The procedure $\text{add_label}(s, fi)$ sets $L[s][fi]$ to true, and the procedure $\text{labeled}(s, fi)$ simply returns the current value of $L[s][fi]$.

In order to handle an arbitrary CTL formula f we successively apply the state labelling algorithm described at the beginning of this section to the subformulas of f , starting with simplest (i.e. highest numbered) and working backwards to f :

```
for fi := length(f) step -1 until 1 do
  label_graph (fi);
```

Since each pass through the loop takes time $O(\text{size}(S) + \text{card}(R))$, we conclude that the entire algorithm requires $O(\text{length}(f) \cdot (\text{card}(S) + \text{card}(R)))$.

3.1 Theorem.

There is an algorithm for determining whether a CTL formula f is true in state s of the structure $M = (S, R, P)$ which runs in time $O(\text{length}(f) \cdot (\text{card}(S) + \text{card}(R)))$. \square

We illustrate the model checking algorithm by considering the global state graph for the solution to the two process *mutual exclusion problem* given in Figure 2.3. In order to establish *absence of starvation* for process 1 we consider the CTL formula $T_1 \rightarrow AF C_1$ or, equivalently, $\neg T_1 \vee AFC_1$. In this case the set of subformulas contains $\neg T_1 \vee AFC_1$, $\neg T_1$, T_1 , AFC_1 and C_1 . The states of the global transition graph will be labeled with these subformulas during execution of the model checking algorithm. On termination every state will be labeled with $\neg T_1 \vee AFC_1$ as shown in Figure 3.1. Thus, we can conclude that $s_0 \models AG(T_1 \rightarrow AFC_1)$. It follows that process 1 cannot be prevented from entering its critical region once it has entered its trying region.

4. Introducing Fairness into CTL

In verifying concurrent systems, we are occasionally interested only in correctness along *fair* execution sequences. For example, with a system of concurrent processes we may wish to consider only those computation sequences in which each process is executed infinitely often. When dealing with network protocols where processes communicate over an imperfect (or lossy) channel we may also wish to restrict the set of computation sequences; in this case the *unfair* execution sequences are those in which a sender process continuously transmits messages without any reaching the receiver due to erratic behavior by the channel.

Roughly speaking, a fairness condition asserts that requests for service are granted "sufficiently often". Different concepts of what constitutes a "request" and what "sufficiently often" should mean give rise to a variety of notions of fairness. Indeed, many different types of fairness and approaches to dealing with them have been proposed in the literature; we refer the reader to [8], [11], [17], and [12] for more extensive treatments.

In this section we will show how to extend the CTL model checking algorithm to handle a simple but fundamental type of fairness in which certain predicates must hold infinitely often along every fair path. In this case it follows from [5] that correctness of fair executions cannot be expressed in CTL. In fact, CTL cannot express the property that some proposition Q should eventually hold on all fair executions.

In order to handle fairness and still obtain an efficient model checking algorithm we modify the semantics of CTL. The new logic, which we call CTL^F , has the same syntax as CTL. But a structure is now a 4-tuple (S, R, P, F) where S, R, P have the same meaning as in the case of CTL, and F is a collection of predicates on S i.e. $F \subseteq 2^S$. A path p is *F-fair* iff the following condition holds: *for each $g \in F$, there are infinitely many states on p which satisfy predicate g* . CTL^F has exactly the same semantics as CTL except that all path quantifiers range over fair paths.

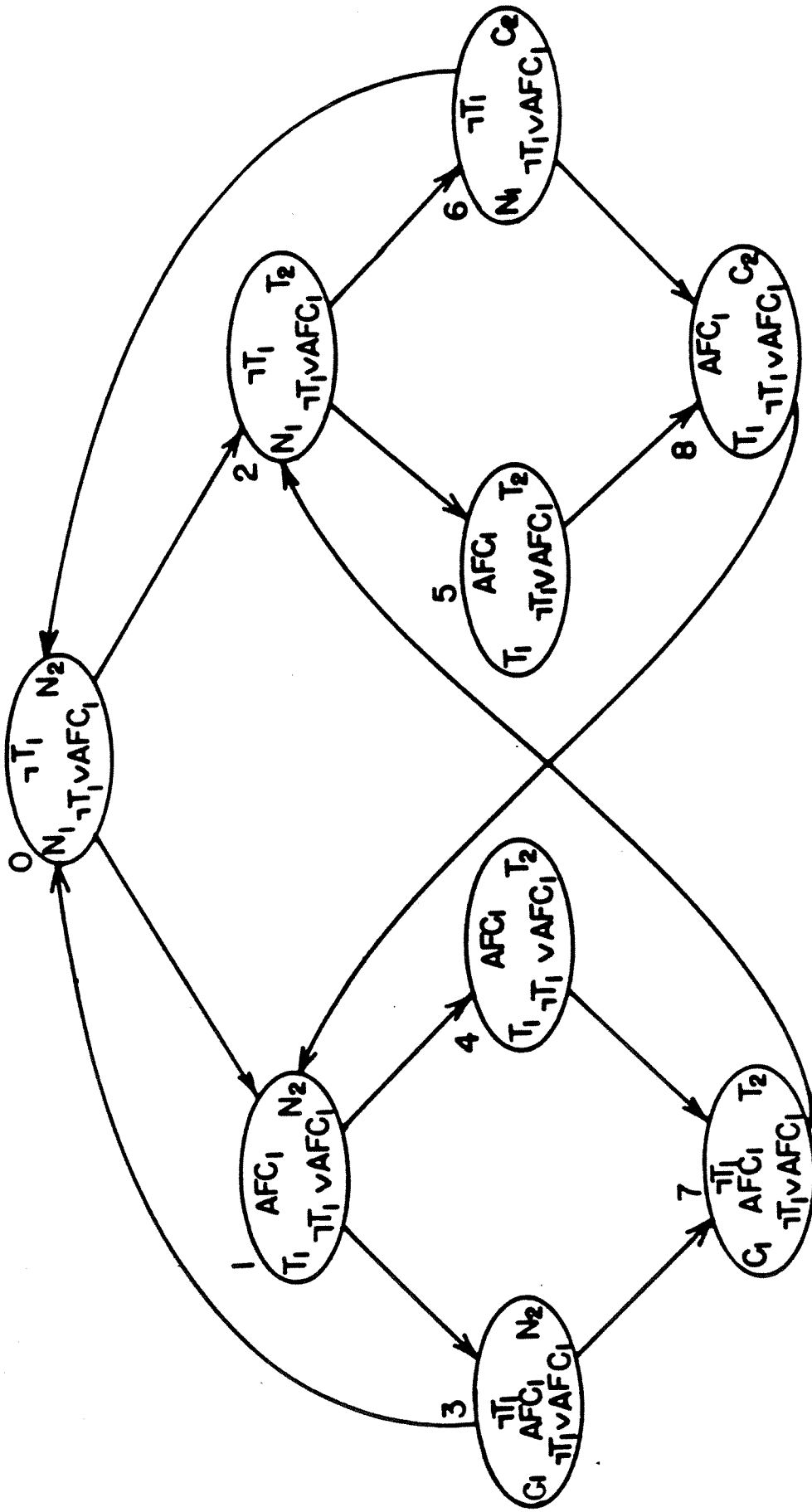


Fig. 3.1 : Global state transition graph after termination of model checking algorithm.

4.1 Lemma.

Given any finite structure $M = (S, R, P)$, collection $F = \{G_1 \dots G_k\}$ of subsets of S , and state $s_0 \in S$ the following two conditions are equivalent:

1. There exists an F -fair path in M starting at s_0 .
2. There exists a strongly connected component C of (the graph of) M such that
 - a. there is a finite path from s_0 to a state $t \in C$, and
 - b. for each G_i , there is a state $t_i \in C \cap G_i$.

Proof:

(1) \Rightarrow (2): Suppose the F -fair path s_0, s_1, s_2, \dots exists in M . Then for each G_i there is a state $t_i \in G_i$ for which there exist infinitely many s_j that are equal to t_i . So for each pair t_i, t_j there is a path (which is some finite segment of the original path) from t_i to t_j . It follows that all the t_i lie in the same strongly connected component C of M . Certainly, there is a path from s_0 to some node $t \in C$ (take $t = t_1$). Moreover, by the choice of the t_i , each $t_i \in C \cap G_i$. Thus, C is the desired strongly connected component of (2).

(2) \Rightarrow (1): Suppose the strongly connected component C exists in M . Then finite paths of the following forms are also present in M : $(s_0, \dots, t_1), (t_1, \dots, t_2), \dots, (t_{k-1}, \dots, t_k)$, and (t_k, \dots, t_1) . We then concatenate these finite paths to get a path: $s_0, \dots, t_1, \dots, t_2, \dots, t_k, \dots, t_1, \dots, t_2, \dots, t_k, \dots, t_1, \dots, t_2, \dots, t_k, \dots$. This path certainly starts at s_0 . Moreover, for each i , there are infinitely many occurrences of $t_i \in G_i$ along it. Thus, this path is F -fair. \square

We next extend our model checking algorithm to CTL^F . We introduce an additional proposition Q , which is true at a state iff there is a fair path starting from that state. This can easily be done, by obtaining the strongly connected components of the graph denoted by the structure. A strongly connected component is *fair* if it contains at least one state from each G_i in F . By the above lemma every state in a fair strongly connected component is the start of an infinite fair path. Thus, we label a state with Q iff there is a path from that state to some node of a fair strongly connected component.

As usual we design the algorithm so that after it terminates each state will be labeled with the subformulas of f_0 true in that state. For checking only fair paths, we consider the two interesting cases where $f \in \text{sub}(f_0)$ and either $f = E[f_1 \cup f_2]$ or $f = A[f_1 \cup f_2]$. We assume that the states have already been labeled with the immediate subformulas of f by an earlier stage of the algorithm.

(i) $f = E[f_1 \cup f_2]$: f is true in a state iff the CTL formula $E[f_1 \cup (f_2 \wedge Q)]$ is true in that state, and this can be determined using the CTL model checker. Note that since fair paths are infinite, the path satisfying f cannot simply stop with the state satisfying f_2 . Again, state s is labeled with f iff f is true in that state.

(ii) $f = A[f_1 \cup f_2]$: It is easy to see that $A[f_1 \cup f_2] = \neg (E[\neg f_2 \cup (\neg f_1 \wedge \neg f_2)] \vee EG(\neg f_2))$. For a state s

we can easily check if $s \models E[\neg f_2 \cup (\neg f_1 \wedge \neg f_2)]$ using the previous technique. To check if $s \models EG(\neg f_2)$ we use the following procedure. Let G_R be the graph corresponding to the above structure. From G_R eliminate all nodes v such that $f_2 \in \text{label}(v)$ and let G_R' be the resultant labeled graph. Find all the strongly connected components of G_R' and mark those which are fair. If s is in G_R' and there is a path from s to a fair strongly component of G_R' then $s \models EG(\neg f_2)$; otherwise $s \models \neg EG(\neg f_2)$. As in (i), s is labeled with f iff f is true in s .

If $n = \max(\text{card}(S), \text{card}(R))$, $m = \text{length}(f)$ and $p = \text{card}(F)$, then it is not difficult to show that the above algorithm takes time $O(n \cdot m \cdot p)$.

An obvious question is whether our approach can handle the various types of fairness that occur in practice. In [12] three different types of fairness properties have been identified as being particularly useful: These are called *impartiality*, *justice*, and *fairness*. We will argue below that the first two of these properties can be handled by the version of the model checker that is described above and currently implemented. We also argue that the third property can be handled by an extension of the above ideas which we have not yet found necessary to implement.

Impartiality requires that every process should be executed infinitely often. To deal with this property we view an execution of a system Pr of concurrent processes as some interleaving of the execution steps of the individual processes. We model a system of processes by a structure (S, R, P) and labeling function $L: R \rightarrow \text{Pr}$, where S is the set of global states of the system, R is the single step execution relation of the system, and for each transition in R , L gives the process which caused the transition. By duplicating each state in S at most $\text{card}(\text{Pr})$ times, we can model the concurrent system by a structure (S^*, R^*, P^*, F) , in which each state in S^* is reached by the execution of at most one process, and F is a partitioning of S^* such that each element in F is the set of states reached by the execution of one process; thus $\text{card}(F) = \text{card}(\text{Pr})$. The fair paths of the above structure correspond exactly to the impartial execution sequences of the system of processes.

A computation is said to be *just* if every process is either infinitely often disabled, or else it is infinitely often executed. Let d_i hold in a state iff process i is not enabled in that state, and let e_i hold in a state iff that state is reached by an execution of process i . It follows that a path is just iff for each process i , the state predicate $(d_i \vee e_i)$ holds infinitely often on the path. Thus, we see that justice can also be directly handled by the version of the model checking algorithm described above.

A computation is *fair* iff for each process, if the process is infinitely often enabled, then it will be infinitely often executed. Our current system does not handle this property; however, it could be easily modified to handle this property. We sketch below the changes that are necessary and refer the reader to see [7] for details.

First, we must once again change our definition of a CTL structure. A structure will now be a 4-tuple (S,R,P,F) where $S,R,$ and P have the same meaning as above; however F will now consist of a collection of *pairs* of the form (p,q) where p,q are predicates. We say that a path is *fair with respect to* (p,q) if whenever p holds infinitely often on the path then q also holds infinitely often on the path. A path is *fair* iff it is fair with respect to every pair (p,q) in F . The semantics of the new logic is the same as CTL except that all path quantifiers range over paths that are fair according to the new definition. The model checking algorithm for CTL^F given earlier in this section can be generalized to handle this notion of fairness.

5. Using the Extended Model Checker to Verify the Alternating Bit Protocol

In this section we consider a more complicated example to illustrate *fair paths* and to show how the Extended Model Checking (EMC) system might actually be used. The example that we have selected is the *Alternating Bit Protocol* (ABP) originally proposed in [2]. Proofs of correctness of this protocol have been constructed manually in ([9], [11]). We show, instead, how the EMC system can be used to verify properties of this protocol automatically. The algorithm consists of two processes, a *Sender process* and a *Receiver process*, which alternately exchange messages. We will assume (as in [16]) that messages from the Sender to the Receiver are *data messages* and that messages from the Receiver to the Sender are *acknowledgments*. We will further assume that each message is encoded so that garbled messages can be detected. Lost messages will be detected by using time-outs and will be treated in exactly the same manner as garbled messages (i.e. as erroneous messages).

Ensuring that each transmitted message is correctly received can be tricky. For example, the acknowledgment to a message may be lost. In this case the Sender has no choice but to resend the original message. The Receiver must realize that the next data message it receives is a duplicate and should be discarded. Additional complications may arise if this message is also garbled or lost. These problems are handled in the algorithm of [2] by including with each message a control bit called the *alternation bit*.

In the EMC system finite-state concurrent programs are specified in a restricted subset of the CSP programming language [10] in which only boolean data types are permitted and all messages between processes must be *signals*. CSP programs for the Sender and Receiver processes in the ABP are shown Appendix II. To simulate garbled or lost messages we systematically replace each message transmission statement by a (nondeterministic) alternative statement that can potentially send an error message instead of the original message. Thus, for example, Receiver ! mess0 would be replaced by

$$\begin{array}{l} [\text{True} \rightarrow \text{Receiver ! mess0} \\ \quad \square \\ \text{True} \rightarrow \text{Receiver ! err}] \end{array}$$

A global state graph is generated from the state machines of the individual CSP processes by considering all possible ways in which the transitions of the individual processes may be interleaved. Since construction of the global state graph is proportional to the product of the sizes of the state machines for the individual processes, a simple (correctness preserving) state minimization algorithm is employed to reduce the number of states in the graph. Explicit construction of the global state machine can be avoided to save space by dynamically recomputing the successors of the current state. The global state graph for our version of the ABP has 251 states.

Once the global state graph has been constructed, the algorithm of section 4 can be used to determine if the program satisfies its specifications. In the case of the ABP we require that every data message that is generated by the Sender process is eventually accepted by the Receiver process:

1. $AG (RcvMsg \rightarrow A[RcvMsg \cup (\neg RcvMsg \ \& \ A[\neg RcvMsg \cup SndMsg])])$
2. $AG (SndMsg \ \& \ Smsg \rightarrow A[SndMsg \cup (\neg SndMsg \ \& \ A[\neg SndMsg \cup RcvMsg \ \& \ Rmsg])])$
3. $AG (SndMsg \ \& \ \neg Smsg \rightarrow A[SndMsg \cup (\neg SndMsg \ \& \ A[\neg SndMsg \cup RcvMsg \ \& \ \neg Rmsg])])$.

The formulas imply that sending a message (SndMsg) strictly alternates with receiving a message (RcvMsg) and that if a 0-message (1-message) is sent then a 0-message (1-message) is received. The conjunction of the formulas is not true of the global state graph obtained from the ABP because of infinite paths on which a message is lost or garbled each time that it is retransmitted. For this reason, we consider only those fair paths on which the initial state occurs infinitely often. With this restriction the algorithm of section 4 will correctly determine that the state graph of the ABP satisfies its specification. See Appendix III.

The EMC system is written in a combination of Lisp and C and has been fully operational since January of 1982. Recently a counterexample facility has been added. When the model checker determines that a formula is false, it will attempt to find a path in the graph which demonstrates that the negation of the formula is true. For instance, if the formula has the form $AG(f)$, our system will produce a path to a state in which $\neg f$ holds. This feature is quite useful for debugging purposes.

6. Extended Logics

In this section we consider logics that are more expressive than CTL and investigate their usefulness for automatic verification of finite state concurrent programs. CTL severely restricts the type of formula that can appear after a path quantifier--only single linear time operator F, G, X, or U can follow a path quantifier. We consider several natural ways of relaxing this restriction. In each case, we will see that the resulting logic has a

model checking problem of intractable complexity (assuming P does not equal NP). We believe that this justifies our decision to restrict our attention to CTL and CTL^F.

The first logic, CTL*, permits an arbitrary formula of linear time logic to follow a path quantifier. We distinguish two types of formulas in giving the syntax of CTL*: state formulas and path formulas. Any state formula is a CTL* formula.

$$\langle \text{state-formula} \rangle ::= \langle \text{atomic proposition} \rangle \mid \langle \text{state-formula} \rangle \wedge \langle \text{state-formula} \rangle \mid \neg \langle \text{state-formula} \rangle \mid E(\langle \text{path-formula} \rangle)$$

$$\langle \text{path-formula} \rangle ::= \langle \text{state-formula} \rangle \mid \langle \text{path-formula} \rangle \cup \langle \text{path-formula} \rangle \mid \neg \langle \text{path-formula} \rangle \mid \langle \text{path-formula} \rangle \wedge \langle \text{path-formula} \rangle \mid X \langle \text{path-formula} \rangle \mid F \langle \text{path-formula} \rangle$$

We use the abbreviation Gf for $\neg F \neg f$ and A(f) for $\neg E \neg (f)$. We interpret state formulas over states of a structure and path formulas over paths of a structure in a natural way. A formula of the form E($\langle \text{path formula} \rangle$) is true in a state iff there is a path in the structure starting from that state on which the path formula is true. The truth of a path formula is defined in much the same way as for a formula in linear temporal logic if we consider all the immediate state subformulas as atomic propositions [5].

More precisely, let $M=(S,R,P)$ be a structure and $p=(s_0, s_1, \dots)$ denote a path in M; $p^{(i)}$ will represent the the suffix of p starting at s_i .

The truth of a *state formula* is defined with respect to a state of M: $s \models E(\langle \text{path formula} \rangle)$ iff there exists a path p in M starting from s such that $\langle \text{path formula} \rangle$ holds at the beginning of the path, i.e., $p \models \langle \text{path formula} \rangle$. A state formula of the form A($\langle \text{path formula} \rangle$) is treated similarly.

The truth of a *path formula* is defined with respect to a path in M; for example, if the path formula is $f_1 U f_2$ we require that $p \models f_1 U f_2$ iff there exist an $i \geq 0$ such that $p^{(i)} \models f_2$ and for all j such that $0 \leq j < i$, $p^{(j)} \models f_1$. If the path formula is a state formula, then we require that $p \models \langle \text{state formula} \rangle$ iff $s_0 \models \langle \text{state formula} \rangle$, where s_0 is the first state on p. The other cases are similar and will be omitted.

BT* will denote the subset of the above logic in which path formulas only use the F operator. CTL⁺ will denote the subset in which the temporal operators X, U, F are not nested.

Fairness can be easily handled in CTL*. For example, the following formula asserts that on all impartial executions of a concurrent system with n processes, R eventually holds:

$$A((GFP_1 \wedge GFP_2 \wedge \dots \wedge GFP_n) \rightarrow FR)$$

Here P_1, P_2, \dots, P_n hold in a state iff that state is reached by execution of one step of process P_1, P_2, \dots, P_n , respectively.

6.1 Theorem.

The model checking problem for CTL is PSPACE-complete.*

Proof Sketch: We wish to determine if the CTL* formula f is true in state s of structure M . Let g be a subformula of f of the form $E(g')$ where g' is a path formula not containing any path quantifiers. For each such g we introduce an atomic proposition Q_g . Let f' be the formula obtained by replacing each such subformula g in f by Q_g . We modify M by introducing the extra atomic-propositions Q_g . Each Q_g is true in a state of the modified structure iff g is true in the corresponding state in M . The latter problem can be solved in polynomial space using the algorithm given in [18]. f is true at state s in M iff f' is true in state s in the modified structure. We successively repeat the above procedure, each time reducing the depth of nesting of the path quantifiers. It is easily seen that the above procedure takes polynomial space. Model checking for CTL* is PSPACE-hard because model checking for formulas of the form $E(g')$, where g' is free of path quantifiers, is shown to be PSPACE-hard in [18]. \square

6.2 Theorem.

The model checking problem for BT^ (and also for CTL^+) is both NP-hard and co-NP-hard and is in Δ_2^P .*

Proof Sketch: The lower bounds follow from the results in [18]. In [18] it was shown that the model checking problem for formulas of the form $F(g)$, where g is free of path quantifiers and uses the only temporal operator F , is in NP. Using this result and a procedure like the one in the proof of previous theorem it is easily seen that the model checking problem for BT^* is in Δ_2^P . A similar argument can be given for CTL^+ . \square

7. Discussion

Much research in protocol verification has attempted to exploit the fact that protocols are frequently finite state. For example, in [19] (global-state) *reachability tree* constructions are described which permit mechanical detection of system deadlocks, unspecified message receptions, and non-executable process interactions in finite-state protocols. An obvious advantage that our approach has over such methods is flexibility; our use of temporal logic provides a uniform notation for expressing a wide variety of correctness properties. Furthermore, it is unnecessary to formulate protocol specifications as reachability assertions since the model checker can handle both safety and liveness properties with equal facility.

The use of temporal logic for specifying concurrent systems has, of course, been extensively investigated ([13], [9], [15]). However, most of this work requires that a proof be constructed in order to show that a

program actually meets its specification. Although this approach can, in principle, avoid the construction of a global state machine, it is usually necessary to consider a large number of possible process interactions when establishing non-interference of processes. The possibility of automatically synthesizing finite state concurrent systems from temporal logic specifications has been considered in [6] and [14], but the synthesis algorithms have exponential-time complexity in the worst case.

Perhaps the research that is most closely related to our own is that of Quielle and Sifakis ([16], [17]), who have independently developed a system which will automatically check that a finite state CSP program satisfies a specification in temporal logic. The logical system that is used in [16] is not as expressive as CTL, however, and no attempt is made to handle fairness properties. Although fairness is discussed in [17], the approach that is used is much different from the one that we have adopted. Special temporal operators are introduced for asserting that a property must hold on fair paths, but neither a complexity analysis nor an efficient model checking algorithm is given for the extended logic.

Acknowledgment

The authors wish to acknowledge the help of M. Brinn, K. Sorenson, and David Dill in implementing an experimental prototype of the system described in section 5.

Appendix I

To establish the correctness of the state labeling in section 3 algorithm we must show that

$$\forall s \text{ [labeled}(s, f) \leftrightarrow s \models f]$$

holds on termination. Without loss of generality we consider only the case in which f has the form $A[f_1 \cup f_2]$. We further assume that the states are already correctly labeled with the subformulas f_1 and f_2 . The first step in the proof is an induction on depth of recursion for the procedure au . Let I be the conjunction of the following eight assertions:

- I1. All states are correctly labeled with the subformulas f_1 and f_2 :
 $\forall s \text{ [labeled}(s, f_i) \leftrightarrow s \models f_i]$ for $i = 1, 2$.
- I2. The states on the stack form a path in the state graph:
 $\{\forall i \text{ [} 1 \leq i < \text{length}(\text{ST}) \rightarrow (\text{ST}(i), \text{ST}(i+1)) \in R \text{]}\}$
- I3. The current state parameter of au is a descendant of the state on top of the stack:
 $(\text{Top}(\text{ST}), s) \in R$.
- I4. $f_1 \wedge \neg f_2$ holds at each state on the stack :
 $\forall i \text{ [} 1 \leq i \leq \text{length}(\text{ST}) \rightarrow \text{ST}(i) \models f_1 \wedge \neg f_2 \text{]}$.
- I5. Every state on the stack is marked but unlabeled :
 $\forall i \text{ [} 1 \leq i \leq \text{length}(\text{ST}) \rightarrow \text{marked}(\text{ST}(i)) \wedge \neg \text{labeled}(\text{ST}(i), f) \text{]}$.
- I6. If a state is labeled with f , then it also marked and f is true in that state:
 $\forall s \text{ [labeled}(s, f) \rightarrow \text{marked}(s) \wedge s \models f]$.
- I7. If a state is marked but neither labeled with f nor on the stack, then f must be false in that state:
 $\forall s \text{ [marked}(s) \wedge \neg \text{labeled}(s, f) \wedge \neg \exists i \text{ [} 1 \leq i \leq \text{length}(\text{ST}) \wedge s = \text{ST}(i) \text{] } \rightarrow s \models \neg f]$.
- I8. ST_0 records the contents of the stack before the call on au . The final value of ST after the call on procedure au must be equal to the original value before the call: $\text{ST} = \text{ST}_0$.

We claim that if I holds before execution of $au(f, s, b)$, then I will also hold on termination of au ; Moreover, the boolean result parameter b will be true iff f holds in state s . In the standard Hoare triple notation for partial correctness assertions the inductive hypothesis would be

$$\{I\} au(f, s, b) \{I \wedge (b \leftrightarrow s \models f)\}.$$

Once the inductive hypothesis is proved, the correctness of our algorithm is easily established. If the stack is empty before the call on au , we can deduce that both of the following conditions must hold:

- a. $\forall s \text{ [marked}(s) \rightarrow \text{[labeled}(s, f) \rightarrow s \models f \text{]]}$ (from I6).
- b. $\forall s \text{ [marked}(s) \rightarrow \text{[}\neg\text{labeled}(s, f) \rightarrow s \models \neg f \text{]]}$ (from I7, I8).

It follows that

$$\forall s \text{ [marked}(s) \rightarrow \text{[labeled}(s, f) \leftrightarrow s \models f \text{]]}.$$

Because of the **for** loop L in the calling program for au , every state will eventually be marked. Thus, when loop L terminates $\forall s \text{ [labeled}(s, f) \leftrightarrow s \models f \text{]}$ must hold.

Proof of the inductive hypothesis is straightforward but tedious and will be left to the reader. The only tricky case occurs when the state s is marked and on the stack. In this case procedure au simply sets b to false and returns. To see that this is the correct action, we make use of the following observation:

3.2 Lemma:

Suppose there exists a path $(s_1, s_2, \dots, s_m, s_k)$ in the state graph such that $1 \leq k \leq m$ and $\forall i \text{ [} 1 \leq i \leq m \rightarrow s_i \models \neg f_2 \text{]}$, then $s_k \models \neg A[f_1 \cup f_2]$. \square

Appendix II (Alternating Bit Protocol)

```

-- Alternating Bit Protocol
--
-- Variables:
--   exit1 - A bit has been sent and acknowledged.
--   exit2 - A bit has been received.
--   Smsg  - The bit that was sent.
--   Rmsg  - The bit that was received.
-- Labels:
--   SndMsg - The previous message has been acknowledged and a new bit
--             is ready to be sent.
--   RcvMsg - A bit has just been received and the acknowledgement is
--             ready to be sent.
-- Signals:
--   dmXY - Used to send bit X with control bit Y.
--   amX  - Used to acknowledge a bit with control bit X.
--   err  - Used to indicate a scrambled message.
--
AB :: [
  exit1, exit2, Smsg, Rmsg: bool;
  SndMsg, RcvMsg: label;
  dm00, dm01, dm10, dm11, err, am0, am1: signal;
  [
    SND, RCV: process;
    --
    -- Sending process
    --
    SND
  ||
    --
    -- Receiving process
    --
    RCV
  ]
]

```


Receiving Process

```

RCV :: [ *[ true ->
  exit2 := false;
  -- Wait for a data message with control bit 0.
  -- If any other message is received, retransmit
  -- the acknowledgement of the last message (am1).
  *[ ~exit2 -> [ SND ? dm10 -> exit2 := true;
                Rmsg := true
                []
                SND ? dm00 -> exit2 := true;
                Rmsg := false
                []
                SND ? dm11 -> SND ! am1
                []
                SND ? dm01 -> SND ! am1
                []
                SND ? err -> SND ! am1
            ]
  ];
  <<RcvMsg>>
  -- Send an acknowledgement. At this point,
  -- Rmsg contains the bit that was transmitted.
  SND ! am0;
  exit2 := false;
  -- Wait for a data message with control bit 1.
  -- If any other message is received, retransmit
  -- the acknowledgement of the last message (am0).
  *[ ~exit2 -> [ SND ? dm11 -> exit2 := true;
                Rmsg := true
                []
                SND ? dm01 -> exit2 := true;
                Rmsg := false
                []
                SND ? dm10 -> SND ! am0
                []
                SND ? dm00 -> SND ! am0
                []
                SND ? err -> SND ! am0
            ]
  ];
  <<RcvMsg>>
  -- Send an acknowledgement. At this point,
  -- Rmsg contains the bit that was transmitted.
  SND ! am1
] ]

```

Appendix III (Transcript of Model Checker Execution)

{Time is measured in 1/60 of a second. The first component measures total user cpu time. The second component measures total system cpu time.}

```
% emc altbit.1
          CTL MODEL CHECKER (C version 2.0)

Taking input from altbit.1...
Fairness constraint: .

time: (316 32)

|= AG (RcvMsg -> A[RcvMsg U (~RcvMsg & A[~RcvMsg U SndMsg])]).
The equation is FALSE.

time: (399 44)

|= AG (SndMsg & Smsg -> A[SndMsg U (~SndMsg & A[~SndMsg U RcvMsg & Rmsg])]).
The equation is FALSE.

time: (469 60)

|= AG (SndMsg & ~Smsg-> A[SndMsg U (~SndMsg & A[~SndMsg U RcvMsg & ~Rmsg])]).
The equation is FALSE.

time: (529 72)

|= (restart)
Fairness constraint: SndMsg.
Fairness constraint: RcvMsg.
Fairness constraint: .

time: (563 76)

|= AG (RcvMsg -> A[RcvMsg U (~RcvMsg & A[~RcvMsg U SndMsg])]).
The equation is TRUE.

time: (595 79)

|= AG (SndMsg & Smsg -> A[SndMsg U (~SndMsg & A[~SndMsg U RcvMsg & Rmsg])]).
The equation is TRUE.

time: (643 81)

|= AG (SndMsg & ~Smsg-> A[SndMsg U (~SndMsg & A[~SndMsg U RcvMsg & ~Rmsg])]).
The equation is TRUE.

time: (694 83)

|= .
End of Session.
```

References

1. M. Ben-Ari, A. Pnueli, Z. Manna. "The Temporal Logic of Branching Time". *Acta Informatica* 20 (1983), 207-226.
2. K.A. Bartlet, R.A. Scantlebury, P.T. Wilkinson. "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links". *Communications of the ACM* 12, 5 (1969), 260-261.
3. E.M. Clarke, E.A. Emerson. "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic". Proceedings of the Workshop on Logic of Programs, Yorktown-Heights, NY, Lecture Notes in Computer Science # 131, 1981.
4. E.A. Emerson, E.M. Clarke. "Characterizing Properties of Parallel Programs as Fixpoints". Proceedings of the Seventh International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science #85, 1981.
5. E.A. Emerson, J.Y. Halpern. ""Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic". Proceedings of the ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, Austin, Texas, January, 1982. to appear in JACM.
6. E. A. Emerson, E. M. Clarke. "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons". *Science of Computer Programming* 2 (1982), 241-266".
7. E.A. Emerson, Chin Laung Lei. Modalities for Model Checking: Branching Time Strikes Back. Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New Orleans, January, 1985, pp. 84-95.
8. D. Gabbay, A. Pnueli, S. Shelah, J. Stavi. The Temporal Analysis of Fairness. Seventh ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, Las Vegas, January, 1980, pp. 163-173.
9. B.T. Hailpern. *Lecture Notes in Computer Science*. Volume 129: "Verifying Concurrent Processes Using Temporal Logic". Springer Verlag, 1982. Ph.D. Thesis, Stanford University, 1980.
10. C.A.R. Hoare. "Communicating Sequential Processes". *Communications of the ACM* 21, 8 (August 1978), 666-667.
11. Leslie Lamport. "Sometimes" is Sometimes "Not Never". Seventh Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, Las Vegas, January, 1980, pp. 174-185.
12. D. Lehmann, A. Pnueli, J. Stavi. "Impartiality, Justice, and Fairness: The Ethics of Concurrent Termination". *Automata, Languages, and Programming, Springer Verlag LNCS 115* (1981), 265-277.
13. Z. Manna, A. Pnueli. Verification of Concurrent Programs: The Temporal Framework. In *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, Ed., Academic Press, London, 1981, pp. 215-273.
14. Z. Manna, P. Wolper. "Synthesis of Communicating Processes from Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems* 6 (1984), 68-93.
15. S. Owicki, L. Lamport. "Proving Liveness Properties of Concurrent Programs". *ACM Transactions on Programming Languages and Systems* 4, 3 (July 1982), 455-495.

16. J.P. Quielle, J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". Proceedings of the Fifth International Symposium in Programming, 1981, pp. 337-350. Springer Lecture Notes in Computer Science 137.
17. J.P. Quielle, J. Sifakis. Fairness and Related Properties in Transition Systems. 292, IMAG (University of Grenoble), March, 1982.
18. A.P. Sistla, E.M. Clarke. "Complexity of Propositional Linear Temporal Logics" . *Journal of the Association for Computing Machinery* 32, 3 (July 1985), 733-749.
19. P. Zafiropulo, C. West, H. Rudin, D. Cowan, D. Brand. "Towards Analyzing and Synthesizing Protocols". *IEEE Transactions on Communications COM-28*, 4 (April 1980), 651-671.