Automatic Verification Of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach*

E.M. Clarke
Carnegie-Mellon University
E.A. Emerson
University of Texas, Austin
A.P. Sistla
Harvard University

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 transition graph for the concurrent system. We also show how the logic and our algorithm can be modified to handle *fairness*. We argue that this technique can provide a practical alternative to manual proof construction or use of a mechanical theorem prover for verifying many finite state concurrent systems.

1. Introduction.

In the traditional approach to concurrent program verification, the proof that a program meets its specifications is constructed by hand using various axioms and inference rules in a deductive system such as temporal logic ([8], [6], [10]). 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 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 given 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 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.

*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 and a departmental grant from IBM.

2. The Specification Language.

The 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 formulae, 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 formulae 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 formulae 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 is an assignment of atomic propositions to states i.e. $P : S \rightarrow 2^{AP}$.

A *path* is an infinite sequence of states $(s_0, s_1, s_2, ...)$ 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 $(s,t) \in R$.

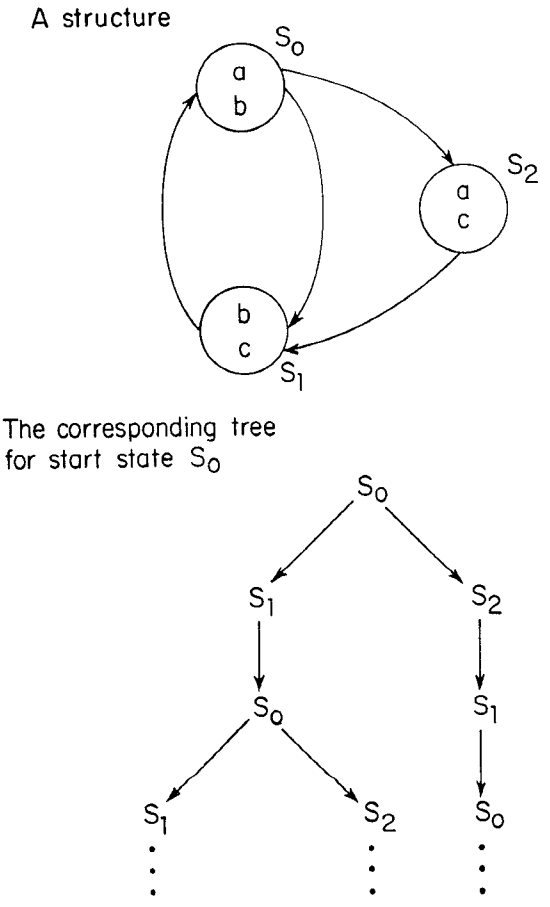

Figure 2.1

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:

$s_0 \models p$ iff $p \in P(s_0)$.

$s_0 \models \neg f$ iff not($s_0 \models f$).

$s_0 \models f_1 \wedge f_2$ iff $s_0 \models f_1$ and $s_0 \models f_2$.

$s_0 \models AXf_1$ iff for all states t such that $(s_0,t) \in R$, $t \models f_1$.

$s_0 \models EXf_1$ iff for some state t such that $(s_0,t) \in R$, $t \models f_1$.

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

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

3. Model Checker

Assume that we wish to determine whether formula f_o is true in the finite structure M = (S, R, P). We design our algorithm so that when it finishes, each state will be labelled with the set of subformulae true in the state. We let label(s) denote this set for state s. Consequently, M, $s \models f$ iff $f \in$ label(s) at termination. In order to explain our algorithm we first consider the case in which each state is currently labelled with the **immediate** subformulae of f which are true in that state.

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

- arg1(f) and arg2(f) give the first and second arguments of a two argument formula f such as $A[f_1 U f_2]$.
- labelled (s, f) will return true (false) if state s is (is not) labelled with formula f.
- add_label(s, f) adds formula f to the current label of state s.

Our state labelling algorithm (**procedure** 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, $A[f_1 U f_2]$, or $E[f_1 U f_2]$. We will only consider the case in which $f = A[f_1 U f_2]$ here since all of the other cases are either straightforward or similar. For the case $f = A[f_1 U f_2]$ our algorithm uses a depth first search to explore the state graph. The bit array marked[1: nstates] is used to indicate which states have been visited by the search algorithm. The algorithm also uses a

stack ST to keep track of those states which require additional processing before the truth or falsify of f can be determined. The boolean procedure stacked(s) will determine (in constant true) whether state s is currently on the stack ST.

```
begin
    ST := empty_stack;
    for all s ∈ S do marked(s) := false;
    L : for all s ∈ S do
    if ¬ marked(s) then au(f,s,b)
end
```

The recursive procedure 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 ⊨ f. The annotated code for procedure au is shown below: .

```
procedure au(f,s,b)
begin

    {If s is marked and stacked, return false (see lemma 3.1).
    If s is already labelled with f, then return true. Otherwise,
    if s is marked but neither stacked nor labelled, then
    return false.}

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

    {Mark state s as visited. Let f = A[f1 U f2]. If f2 is true at
    s, f is true at s; so label s with f and return true. If f1 is not
    true at s, then f is not true at s; so return false. }

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

    {Push s on stack ST. Check to see if f is true at all
    successor states of s. If there is some successor state s1 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.}

    push(s,ST);
    for all s1 ∈ successors(s) do
        begin
            au (f,s1,b1);
            if ¬b1 then
                begin
                    pop(ST);
                    b := false;
                    return
                end
        end;
    pop(ST);
    add_label(s,f);
    b := true;
    return
end of procedure au.
```

To establish the correctness of the algorithm we must show that

$$\forall s \left[\text{labelled}(s,f) \leftrightarrow s \models f \right]$$

holds on termination. Without loss of generality we consider only the case in which f has the form $A[f_1 \, U \, f_2]$. We further assume that the states are already correctly labelled with the subformulae 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 labelled with the subformulae f_1 and f_2: $\forall s \left[\text{labelled}(s,f_i) \leftrightarrow s \models f_i \right]$ for i = 1,2.

I2. The states on the stack form a path in the state graph: $\forall i \left[1 \leq i < \text{length}(ST) \rightarrow (ST(i), ST(i+1)) \in R \right]$.

I3. The current state parameter of au is a descendant of the state on top of the stack: $(\text{Top}(ST), s) \in R$.

I4. $f1 \wedge \neg f2$ holds at each state on the stack : $\forall i \left[1 \leq i \leq \text{length}(ST) \rightarrow ST(i) \models f1 \wedge \neg f2 \right]$.

I5. Every state on the stack is marked but unlabelled : $\forall i \left[1 \le i \le \text{length}(ST) \rightarrow \text{marked}(ST(i)) \wedge \neg \text{labelled}(ST(i), f) \right]$.

I6. If a state is labelled with f, then it also marked and f is true in that state: $\forall s \left[\text{labelled}(s,f) \rightarrow \text{marked}(s) \wedge s \models f \right]$.

I7. If a state is marked but neither labelled with f nor on the stack, then f must be false in that state: $\forall s \left[\text{marked}(s) \wedge \neg \text{labelled}(s,f) \wedge \neg \exists i \left[1 \le i \le \text{length}(ST) \wedge s = ST[i] \right] \rightarrow s \models \neg f \right]$.

I8. ST_0 records the contents of the stack: $ST = ST_0$.

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

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

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. $\forall s \left[\text{marked}(s) \rightarrow [\text{labelled}(s, f) \rightarrow s \models f] \right]$ (from I1).

b. $\forall s \left[\text{marked}(s) \rightarrow [\neg \text{labelled}(s,f) \rightarrow s \models \neg f \ ] \right]$ (from I2, I3).

It follows that

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

Because of the **for** loop L in the calling program for au, every state will eventually be marked. Thus, when loop L terminates $\forall s \left[\text{labelled}(s, f) \leftrightarrow s \models f \right]$ 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 situation the procedure au simply sets b to false and returns. To see that this is the correct action, we make use of the following observation:

3.1 Lemma:

Suppose there exists a path $(s_1, s_2, \ldots, s_m, s_k)$ *in the state graph such that* $1 \le k \le m$ *and* $\forall i \left[1 \le i \le m \rightarrow s_i \models \neg f_2 \right]$, *then* $s_k \models \neg A[f_1 U f_2]$. □

Assuming that the states of the graph are already correctly labelled with f_1, and f_2, it is easy to see that the above algorithm requires time O(card(S) + 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 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.

We next show how 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 subformulae 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 subformulae 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 + length(f_1) to the subformulae 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 + length(f_1) to the subformulae of f_1 and i + length(f_1) through i + length(f_1) + length(f_2) to the subformulae of f_2. Thus, in one pass through f we can build two arrays nf[1 : length(f)] and sf[1 : length(f)] where nf[i] is the i^{th} subformula of f in the above numbering and sf[i] is the list of the numbers assigned to the immediate subformulae of the i^{th} formula. For example, if f = (AU (NOT X) (OR Y Z)), then nf and sf are given below:

```
nf[1] (AU (NOT X) (OR Y Z))     sf [1]   (2 4)
nf[2] (NOT X)                   sf [2]     (3)
nf[3] X                         sf [3]     nil
nf[4] (OR Y Z)                  sf [4]   (5 6)
nf[5] Y                         sf [5]     nil
nf[6] Z                         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 "labelled" and "add_label". We associate with each state s a bit array L[s] of size length(f). The procedure add_label(s,fi) sets L[s][fi] to true, and the procedure labelled(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(size(S) + card(R)), we conclude that the entire algorithm requires O(length(f) · (card(S) + card(R))).

3.2 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(length(f) · (card(S) + card(R))). □

We illustrate the model checking algorithm by considering a finite state solution to the *mutual exclusion problem* for two

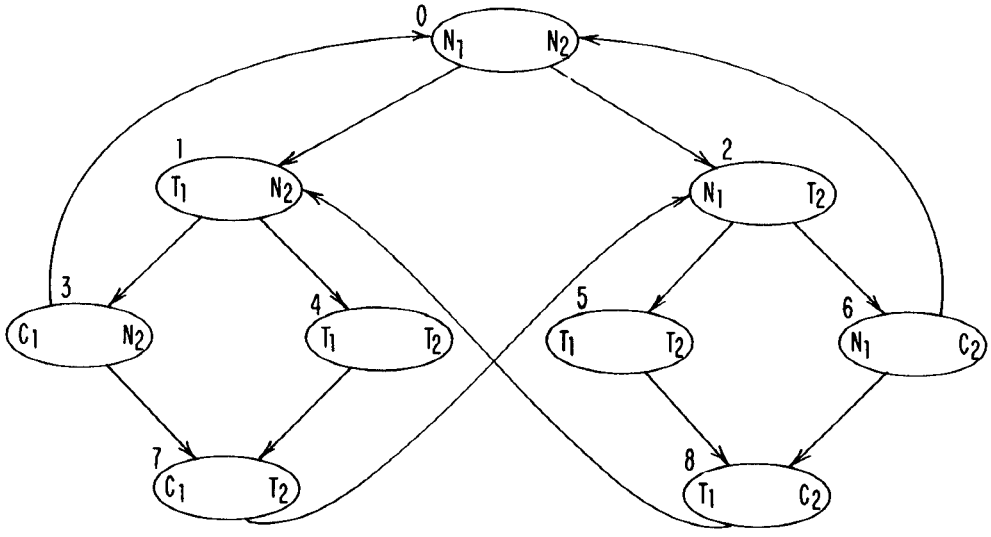

Fig. 3.2a : Global state transition graph for two process mutual exclusion problem.

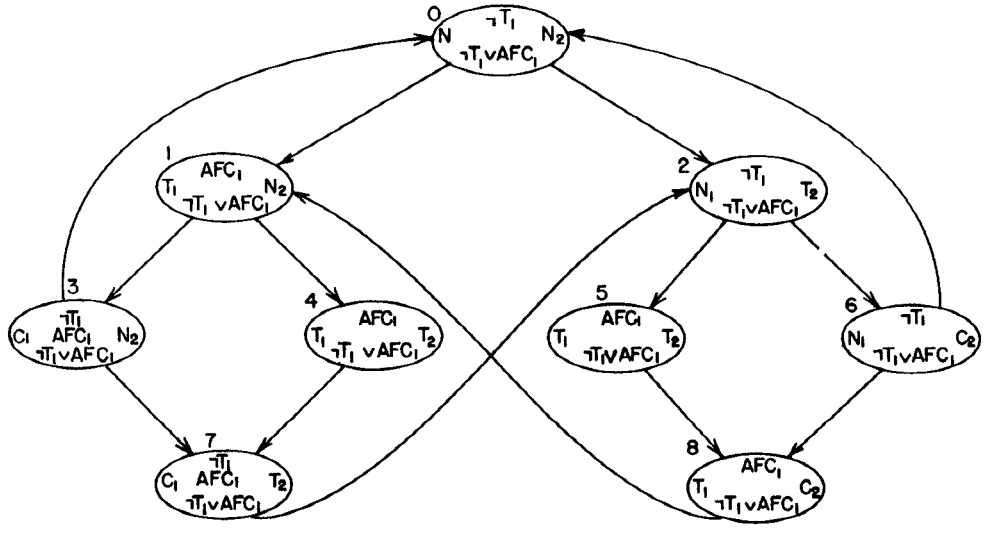

Fig. 3.2 b : Global state transition graph after termination of model checking algorithm.

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.

A *global* state transition graph for this solution is shown in figure 3.1a. 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.

In order to establish *absence of starvation* for process 1 we consider the CTL formula $T_1 \rightarrow AFC_1$ or, equivalently, $\neg T_1 \vee AFC_1$, where $AFp \equiv A[true\ U\ p]$ means that p occurs at some point on all execution paths. In this case the set of subformulae 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 labelled with these subformulae during execution of the model checking algorithm. On termination every state will be labelled with $\neg T_1 \vee AFC_1$ as shown in figure 3.1b. Thus, we can conclude that $s_0 \models AG(T_1 \rightarrow AFC_1)$ where $AGp \equiv \neg E[true\ U\ \neg p]$ means that p holds globally on all computation paths. 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

Frequently, in verifying concurrent systems we are only interested in the correctness of 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 imperfect (or lossy) channels 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. Since we are considering only finite state systems, each of these notions of fairness requires that some collection of states be repeated infinitely often in every fair computation. 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 subsets of S i.e. $F \subseteq 2^S$. A path p is fair iff the following condition holds:

for each $c \in F$, there are infinitely many instances on p at which some state in c appears.

CTL^F has exactly the same semantics as CTL except that all path quantifiers range over fair paths.

An execution of a system Pr of concurrent processes is some interleaving of the execution steps of the individual processes. We can model a system of concurrent processes by a structure (S, R, P) and labelling function $L:R \rightarrow Pr$. 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 card(Pr) times, we can model the concurrent system by a structure (S^*, R^*, P^*, F), where 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 card(F) = card(Pr). The fair paths of the above structure are exactly the fair execution sequences of the system of concurrent processes. A similar approach can be used to model network protocols (see section 5).

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 c_i in F. 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 labelled with the subformulae of f_0 true in that state.

We consider the two interesting cases where $f \in sub(f_0)$ and either $f = E[g\ U\ h]$ or $f = A[g\ U\ h]$. We assume that the states have already been labelled with the immediate subformulae of f by an earlier stage of the algorithm.

(i) $f = E[g\ U\ h]$: f is true in a state iff the CTL formula $E[g\ U\ (h \wedge Q)]$ is true in that state, and this can be determined using the CTL model checker. A state s is labeled with f iff f is true in that state.

(ii) f = A[g U h] : It is easy to see that A[g U h] = ¬ (E[¬h U (¬g ∧ ¬h)] ∨ EG(¬h)). For a state s we can easily check if s ⊨ E[¬h U (¬g ∧ ¬h)] using the previous technique. To check if s ⊨ EG(¬h) 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 h ∈ label(v) and let G_R' be the resultant labeled graph. Find all the strongly connected 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 ⊨ EG(¬h); otherwise s ⊨ ¬ EG(¬h). As in (i), s is labeled with f iff f is true in s.

If n = max(card(S), card(R)), m = length(f) and p = card(F), then it can be shown that the above algorithm takes time $O(n \cdot m \cdot p)$.

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]. This algorithm consists of two processes, a *Sender process* and a *Receiver process*, which alternately exchange messages. We will assume (as in [11]) 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 error 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 [7] 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 in figures 5.1a and 5.1b. 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

[True → Receiver ! mess0
□
True → Receiver ! err]

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, various (correctness preserving) heuristics are 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 the ABP is shown in the figure 5.2.

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:

AG[gen_dm0 → AX[A[¬ (gen_dm0 ∨ gen_dm1) U acc_dm0]] ∧

AG[gen_dm1 → AX[A[¬ (gen_dm0 ∨ gen_dm1) U acc_dm1]]

This formula is not true of the global state graph shown in figure 5.2 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 figure 5.3 satisfies its specification.

As of October 1982, most of the programs that comprise the EMC system have been implemented. The program which parses CSP programs and constructs the global state graph is written in a combination of C and lisp and is operational. An efficient top-down version of the model checking algorithm of section 3 has also been implemented and debugged. The extended model checking algorithm of section 4 (which only considers fair paths) has been implemented in Lisp and is currently being debugged.

(Note: dm stands for data message; am stands for acknowledgement message.)

```
*[  gen_dm0;
    RCV ! dm0;
    *[RCV ? am0 → exit;
    []
    RCV ? aml → RCV ! dm0;
    []
    RCV ? err → RCV ! dm0;
    ]
    gen_dml;
    RCV ! dml;
    *[RCV ? aml → exit;
    []
    RCV ? am0 → RCV ! dml;
    []
    RCV ? err → RCV ! dml;
    ]
 ]
```

Figure 5.1a: Sender Process (SND)

```
*[ *[ SND ? dm0   exit;
      []
      SND ? dml → SND ! aml;
      []
      SND ? err → SND ! aml;
      ]
      acc_dm0;
      SND ! am0;
      *[SND ? dml → exit;
      []
      SND ? dm0 → SND ! am0;
      []
      SND ? err → SND ! am0;
      ]
      acc_dml;
      SND ! aml;
   ]
```

Figure 5.1b: Receiver Process (RCV)

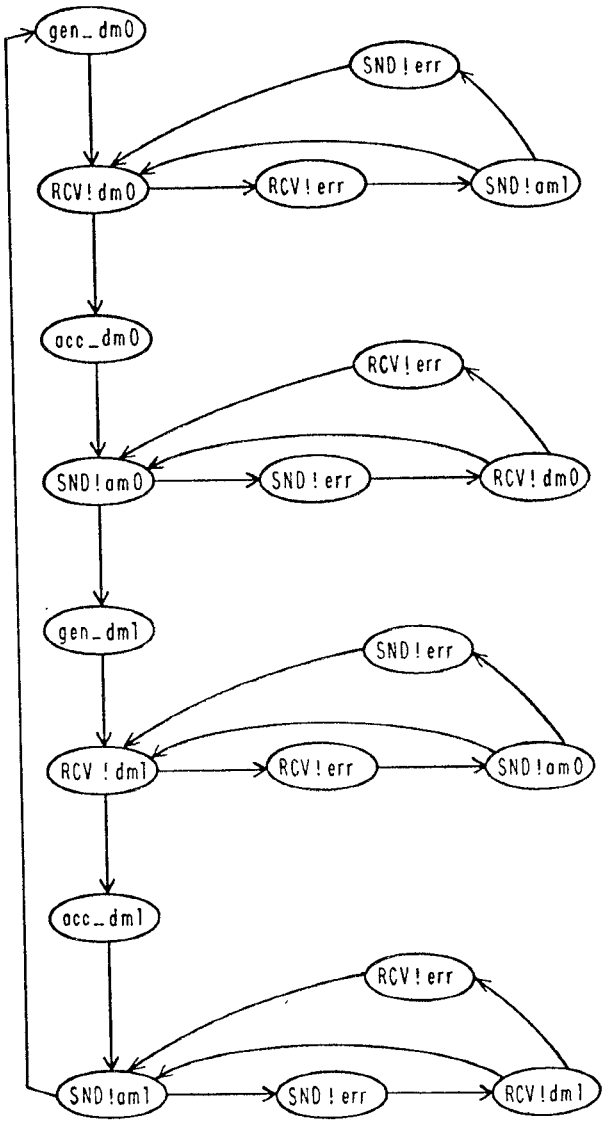

Figure 5.2 Global state transition graph for alternating bit protocol.

6. Extended Logics

In this section we consider logics which are more expressive than CTL and investigate their usefulness for automatic verification of finite state concurrent systems. CTL severely restricts the type of formula that can appear after a path quantifier. In CTL* we relax this restriction and allow an arbitrary formula of linear time logic to follow a path quantifier. We distinguish two types of formulae in giving the syntax of CTL*: state formulae and path formulae. Any state formulae is a CTL* formula.

<state-formula>::= <atomic proposition> |
<state-formula> ∧ <state-formula> |
¬ <state-formula> |
E(<path-formula>)

<path-formula>::= <state-formula> |
<path-formula> U <path-formula> |
¬<path-formula> |
<path-formula> ∧ <path-formula> |
X <path-formula> |
F<path-formula>

We use the abbreviation Gf for ¬F¬f and A(f) for ¬E¬(f). We interpret state formulae over states of a structure and path formulae over paths of a structure in a natural way. The truth of a CTL* formula in a state of a structure is inductively defined. A formula of the form E(<path formula>) 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 - subformulae as atomic propositions [5]. BT^* will denote the subset of the above logic in which path formulae 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 fair executions of a concurrent system with n processes, R eventually holds:

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

Here P_1, P_2,...P_n hold in a state iff that state is reached by execution of one step of process P_1, P_2,...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 [13]. 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 [13]. □

6.2 Theorem.

The model checking problem for BT^* *(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 [13]. In [13] 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^+. □

We believe that the above complexity results justify our approach in section 5 where fairness constraints are incorporated into the semantics of the logic in order to obtain a polynomial-time model checking algorithm.

7. Conclusion

Much research in protocol verification has attempted to exploit the fact that protocols are frequently finite state. For example, in [15] and [14] (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 ([8], [6], [10]). 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 [3] and [9]. But this approach has not been implemented, and 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 ([11], [12]), 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 [11], is not as expressive as CTL, however, and no attempt is made to handle fairness properties. Although fairness is discussed is [12], 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 and K. Sorenson in implementing an experimental prototype of the system described in section 5.

References

1. M. Ben-Ari, Z. Manna, A. Pneuli. "The Logic of Nextime." *Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, VA* (January 1981), 164-176.

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. Synthesis of Synchronization Skeletons for 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. POPL 83

6. B.T. Hailpern, S. Owicki. Verifying Network Protocols Using Temporal Logic. Tech. Rept. 192, Computer System Laboratory, Stanford University, June, 1980.

7. C.A.R. Hoare. "Communicating Sequential Processes ." *Communications of the ACM 21*, 8 (August 1978), 666-667.

8. Z. Manna, A. Pneuli. "Verification of Concurrent Programs: The Temporal Framework." *The Correctness Problem in Computer Science (R.S. Boyer and J.S. Moore, eds.), International Lecture Series in Computer Science* (1981).

9. Z. Manna, P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. Proceedings of the Workshop on Logic of Programs, Yorktown-Heights, NY, 1981.

10. S. Owicki, L. Lamport. "Proving Liveness Properties of Concurrent Programs." *Stanford University Technical Report* (1980).

11. J.P. Quielle, J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. Proceedings of the Fifth International Symposium in Programming, 1981.

12. J.P. Quielle, J. Sifakis. "Fairness and Related Properties in Transition Systems." *IMAG*, 292 (March 1982).

13. A.P. Sistla, E.M. Clarke. "Complexity of Propositional Temporal Logic." (1982).

14. D.P. Sidhu. Rules for Synthesizing Correct Communication Protocols. PNL Preprint, to appear in SIGCOMM

15. 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.