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 ifver, and especially in the case of 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 syntax for CTL is given below. AP is the underlying set of atomic propositions.

1. Every atomic proposition \( p \in \text{AP} \) is a CTL formula.

2. If \( f_1 \) and \( f_2 \) are CTL formulae, then so are \( \neg f_1 \), \( f_1 \land f_2 \), \( A f_1 \), \( E f_1 \), \( A[f_1 \cup f_2] \) and \( E[f_1 \cup f_2] \).

The symbols \( \land \) and \( \neg \) have their usual meanings. \( X \) is the next-time operator; the formulae \( AX f_1 \) (\( EX f_1 \)) intuitively mean that \( f_1 \) holds in every (in some) immediate successor of the current program state. \( U \) is the until operator; the formula \( A[f_1 \cup f_2] \) (\( E[f_1 \cup f_2] \)) intuitively mean that for every computation path (for some computation path), there exists an initial prefix of

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0-89791-090-7/83/001/0117 $00.75
begin
    ST := empty_stack;
    for all s ∈ S do marked(s) := false;
    L := for all s ∈ S do if ¬ marked(s) then au(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 = Afj U fj, If fj is true at s, f is true at s; so label s with f and return true. If fj 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;
    pop(ST);
    add_label(s,f);
    b := true;
    return
end

end of procedure au.

To establish the correctness of the algorithm we must show that

∀s [ labelled(s,f) ⇔ s ⊨ f ]

holds on termination. Without loss of generality we consider only the case in which f has the form Afj U fj. We further assume that the states are already correctly labelled with the subformulae fj and fj. 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:

11. All states are correctly labelled with the subformulae fj and fj: ∀s [ labelled(s,f) ⇔ s ⊨ fj ] for i = 1,2.

12. The states on the stack form a path in the state graph:
∀i [ 1 ≤ k ≤ length(ST) ⇒ (ST(i), ST(i+1)) ∈ R ]

13. The current state parameter of au is a descendant of the state on top of the stack: (Top(ST), s) ∈ R.

14. ⊩ ∀ i, i ≤ length(ST) ⇒ ST(i) = ⊩ ∨ ⊥ holds at each state on the stack:
∀i [ i ≤ length(ST) ⇒ ST(i) = ⊩ ∨ ⊥ ]

110
for \( i = \text{length}(f) \) step -1 until \( i \) do

\[ \text{label}_\text{graph}(i); \]

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

3.2 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 \( O(\text{length}(f) \cdot (\text{card}(S) + \text{card}(R))) \). \( \square \)

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.](image)

![Fig. 3.2b: Global state transition graph after termination of model checking algorithm.](image)
(ii) \( f = A[g \cup h] \): It is easy to see that \( A[g \cup h] = \neg (E^h \neg h \cup (g \land \neg h)) \lor EG(\neg h) \). For a state \( s \) we can easily check if \( s \models E^h \neg h \cup (g \land \neg h) \) using the previous technique. To check if \( s \models EG(\neg h) \) 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 \( h \notin \text{label}(v) \) 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 \( s \models EG(\neg h) \); otherwise \( s \models \neg EG(\neg h) \). As in (i), \( s \) is labeled with \( f \) if \( f \) is true in \( s \).

If \( n = \max(\text{card}(S), \text{card}(R)) \), \( m = \text{length}(f) \) and \( p = \text{card}(F) \), then it 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,

\[
\begin{align*}
\text{Receiver} & \rightarrow \text{Receiver} \oplus \text{mess}0 \\
& \text{True} \\
& \text{True} \rightarrow \text{Receiver} \oplus \text{err}
\end{align*}
\]

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:

\[
AG[\text{gen}_d m_0 \rightarrow AX[A[\neg (\text{gen}_d m_0 \lor \text{gen}_d m_1) \cup \text{acc}_d m_0]]] \\
AG[\text{gen}_d m_1 \rightarrow AX[A[\neg (\text{gen}_d m_0 \lor \text{gen}_d m_1) \cup \text{acc}_d m_1]]]
\]

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.
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.

\[
\langle\text{state-formula}\rangle ::= \langle\text{atomic proposition}\rangle \\
\quad | \langle\text{state-formula}\rangle \land \langle\text{state-formula}\rangle \\
\quad | \langle\text{state-formula}\rangle \\
\quad | E\langle\text{path-formula}\rangle
\]

\[
\langle\text{path-formula}\rangle ::= \langle\text{state-formula}\rangle \\
\quad | \langle\text{path-formula}\rangle \lor \langle\text{path-formula}\rangle \\
\quad | \lnot\langle\text{path-formula}\rangle \\
\quad | E\langle\text{path-formula}\rangle \\
\quad | F\langle\text{path-formula}\rangle \\
\quad | X\langle\text{path-formula}\rangle
\]

We use the abbreviation \(G\) for \(\lnot E\lnot F\) and \(A()\) for \(\lnot E\lnot F\). 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 \(E\langle\text{path-formula}\rangle\) 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 - 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:

\[A((GF_{P_1} \land GF_{P_2} \land \ldots \land GF_{P_n}) \rightarrow FR)\]

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

6.1 Theorem.

The model checking problem for CTL+ is PSPACE-complete. \(\square\)

Proof Sketch: We wish to determine if the CTL* formula \(\phi\) is true in state \(s\) of structure \(M\). Let \(g\) be a subformula of \(\phi\) 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 \(\phi'\) be the formula obtained by replacing each such subformula \(g\) in \(\phi\) 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]. \(\phi\) is true at state \(s\) in \(M\) iff \(\phi'\) 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]. \(\square\)

6.2 Theorem.

The model checking problem for BT+ (CTL*)

is both NP-hard and co-NP-hard, and is in \(\Delta^P_2\). \(\square\)

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 only the 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 \(\Delta^P_2\). A similar argument can be given for CTL+. \(\square\)

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.
Conference Record of the TENTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES

Papers Presented at the Symposium
Austin, Texas
January 24-26, 1983

Sponsored by the
ASSOCIATION FOR COMPUTING MACHINERY
SPECIAL INTEREST GROUP ON AUTOMATA AND COMPUTABILITY THEORY
SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES
Conference Record
of the
TENTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES

Papers Presented at the Symposium
Austin, Texas
January 24-26, 1983

Sponsored by the
ASSOCIATION FOR COMPUTING MACHINERY
SPECIAL INTEREST GROUP ON AUTOMATA AND COMPUTABILITY THEORY
SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES