1986

An improved algorithm for the automatic verification of finite state systems using temporal logic

Michael C. Browne
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
An Improved Algorithm for the Automatic Verification of Finite State Systems Using Temporal Logic

Michael C. Browne
Carnegie-Mellon University
Pittsburgh, Pa. 15213

ABSTRACT: One method of verifying finite state systems is to represent the system as a state-transition graph and use an efficient algorithm, called a model checker, to determine if this structure is a model of a temporal logic specification. An example of this approach was given by Clarke, Emerson and Sistla [3], who developed a model checking algorithm for a propositional branching-time temporal logic called CTL (Computation Tree Logic). Unfortunately, their algorithm assumes that all of the state transitions are unconditional, when in practice the transitions of most finite state machines depend on the values of a set of inputs. In this paper, we present an improved CTL model checking algorithm that allows conditional transitions. Although the worst case complexity of our new algorithm is the same as the original model checker, we show that for certain classes of CTL formulas, the new algorithm is a significant improvement.

1. Introduction

There are two distinct methods of applying temporal logic to system verification. In the first approach, temporal logic is used as a deductive system in which proofs are constructed to show that the system meets its specifications [5, 6]. Although this technique is very powerful, the construction of such proofs can be very difficult for a human and almost impossible for a machine. Furthermore, it is unlikely that any significant degree of automation is possible, since the complexity of most temporal logics is so high that the construction of an automatic theorem prover is probably not feasible.

The other method is model theoretic in nature and is only applicable to finite state systems, such as network protocols and hardware controllers. In this approach, the system is represented as a finite state-transition graph, or Kripke structure. Then, instead of constructing a tedious proof, an efficient algorithm, called a model checker, is used to determine whether or not this structure is a model of the temporal logic specification. Clarke, Emerson and Sistla have presented an example of this approach [3] by developing a model checking algorithm for a propositional branching-time temporal logic called CTL (Computation Tree Logic). The usefulness of this algorithm, which is linear in both the number of states and the size of the specification, has been demonstrated in several other papers [1, 2].

This research was supported by NSF Grant Number MCS-82-16706.
One drawback of this algorithm is that it assumes that the state transitions are *unconditional* in that all of the transitions from a state are always enabled. However, most of the systems encountered in practice (particularly hardware controllers) have state transitions that are enabled or disabled by a set of external input signals. In order to remove these conditional transitions so that this algorithm can be used, it is necessary to replace each state of the original system by a set of states, one for each possible combination of input signals, and to determine which transitions are possible from each new state by evaluating the conditions. If there are \( n \) inputs, there are \( 2^n \) input combinations, so this preprocessing results in an exponential blowup in the number of states. Since the model checker is linear in the number of states, this algorithm is exponential in the number of input signals.

In this paper, we present a new algorithm that can verify systems with conditional transitions without the additional preprocessing that the old algorithm requires. In the worst case, the new algorithm is also exponential in the number of input signals. However, if the formula being verified is in a special class, the new algorithm is linear in the number of input signals and quadratic in the size of the input formula. Since this class of formulas includes all of the formulas that we have ever attempted to verify in practice, we believe that the new algorithm is a significant improvement on the original model checking algorithm.

We have implemented both the original model checker and the new algorithm in C on a VAX/780 in order to compare their performance. We attempted to use both algorithms to verify two hardware controllers (a simple traffic light controller and a UART). In the first test, the old algorithm took 17.8 seconds of CPU time and the new algorithm took 3.3 seconds. In the second test, the new algorithm took 54.5 seconds of CPU time, but the preprocessing for the old algorithm added too many states for our implementation of the old algorithm to handle. The test of the old algorithm was aborted after 2 hours (!).

The paper is organized as follows: In section 2, we describe the syntax and semantics of the logic, called *Computation Tree Logic* (CTL). In section 3, we describe the structure of the finite state systems that we will verify. In section 4, we give an outline of our algorithm. In section 5, we give a detailed proof of part of the algorithm. In section 6, we discuss the efficiency of the algorithm. The paper concludes with a brief discussion about the usefulness of this algorithm and future enhancements.
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 \( \neg f_1, f_1 \land f_2, \ AX f_1, \ EX f_1, \ A[f_1 \ U f_2], \) and \( E[f_1 \ U f_2] \).

The symbols \( \land \) and \( \neg \) have their usual meanings. \( X \) is the next time operator; the formula \( AX f_1 (EX f_1) \) intuitively means that \( f_1 \) holds in every (some) immediate successor of the current 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 Kripke structure is a triple \( M = (S, R, P) \) where

- \( S \) is a finite set of states.
- \( R \) is a binary relation on \( S (R \subseteq S \times S) \) which gives the possible transitions between states and must be total, i.e. \( \forall x \in S \exists y \in S [(x, y) \in R] \).
- \( P \) 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 \( \forall i [(s_i, s_{i+1}) \in R] \).

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:

\[
\begin{align*}
    s_0 \models p & \iff p \in P(s_0). \\
    s_0 \models \neg f & \iff \text{not } (s_0 \models f). \\
    s_0 \models f_1 \land f_2 & \iff s_0 \models f_1 \text{ and } s_0 \models f_2. \\
    s_0 \models AX f_1 & \iff \text{for all states } t \text{ such that } (s_0, t) \in R, t \models f_1. \\
    s_0 \models EX f_1 & \iff \text{for some state } t \text{ such that } (s_0, t) \in R, t \models f_1. \\
    s_0 \models A[f_1 \ U f_2] & \iff \text{for all paths } (s_0, s_1, ...), \exists j \geq 0 [s_i \models f_2 \land \forall 0 \leq i < j [s_i \models f_1]]. \\
    s_0 \models E[f_1 \ U f_2] & \iff \text{for some path } (s_0, s_1, ...), \exists j \geq 0 [s_i \models f_2 \land \forall 0 \leq i < j [s_i \models f_1]].
\end{align*}
\]

We will also use the following abbreviations in writing CTL formulas:
\[ f \lor g = \neg (\neg f \land \neg g) \]

\[ \text{AF } f = A[\text{true } U f] \] intuitively means that \( f \) will hold sometime in the future along every path. (\( f \) is inevitable.)

\[ \text{EF } f = E[\text{true } U f] \] means that \( f \) will hold sometime in the future along some path. (\( f \) is possible.)

\[ \text{AG } f = \neg \text{EF } (\neg f) \] means that \( f \) holds at every state on every path. (\( f \) holds globally.)

\[ \text{EG } f = \neg \text{AF } (\neg f) \] means that there is a path along which \( f \) holds at every state.

In verifying finite state systems, we are often interested only in the system’s behavior along fair paths. For example, if the system allocates a shared resource between several users, we might want to consider only those paths along which no user keeps the resource forever. Unfortunately, it has been shown that CTL cannot express assertions about correctness along fair paths [4]. Therefore, it is necessary to modify the semantics of CTL in order to deal with fairness.

Let \( F \) be a finite set of CTL formulas, \( \{c_0, c_1, \ldots, c_n\} \), called fairness constraints. We define a fair path to be a path along which each fairness constraint is satisfied infinitely often. \( \text{CTL}^F \) (CTL with fairness) has the same syntax as CTL, but the semantics are different in that all of the path quantifiers range over fair paths. For example, the definition of \( \models_F \) (satisfiability in \( \text{CTL}^F \)) for EX is:

\[ s_0 \models \text{EX } f \iff \text{for some fair path } (s_0, s_1, \ldots), s_1 \models f. \]

3. The Finite State System Description

The finite state systems that our algorithm deals with are 5-tuples, \( M = (S, I, O, R, P) \) where

- \( S \) is a finite set of states.
- \( I \) is a finite set of input propositions that transitions can depend on. \( 2^I \) is the finite set of possible input assignments. Let \( L(I) \) be the set of formulas of propositional logic that only contain propositions from \( I \). Let \( \text{sat} \) be the satisfiability relation for \( L(I) \), so that \( \text{A sat } G \) is true if \( A \in 2^I \) satisfies \( G \in L(I) \).
- \( O \) is a finite set of output propositions that label states. \( I \cap O = \emptyset \).
- \( R \) is a set of triples \( (s, t, G) \) from \( S \times S \times \{L(I)\}(\{false\}) \). For any pair of states, \( s \) and \( t \), only one triple \( (s, t, G) \) will be in \( R \). The meaning of a triple is that if the structure is in state \( s \), and the current input state satisfies \( G \) then the structure may make a transition to state \( t \). Every state must have a successor, so \( R \) must satisfy

\[ \forall x \in S \quad \forall A \in 2^I \quad \exists (x, y, G) \in R \quad [A \text{ sat } G]. \]
- \( P \) is an assignment of output propositions to states i.e. \( P : S \rightarrow 2^O \).
Let \( AP \), the set of atomic propositions, be \( I \cup O \). Now, it is necessary to define what it means for a CTL formula to be true in such a structure. For any 5-tuple of this form, we can define an equivalent Kripke structure, \( M' = (S', R', P') \) where

- \( S' = \{ \langle s, A \rangle | s \in S & A \in 2^I \} \).
- \( R' = \{ \langle \langle s, A \rangle, \langle t, B \rangle \rangle | \langle s, A \rangle, \langle t, B \rangle \in S' & \langle s, t, G \rangle \in R & A \text{ sat } G \} \).
- \( P'(<s, A>) = P(s) \cup A \).

We can apply the CTL semantics in section 2 to define truth in \( M' \). Therefore, we can define truth in \( M \) at state \( s \) with input state \( A \) by the equivalence:

\[ M, s, A \models f \iff M', \langle s, A \rangle \models f. \]

We can define \( f \) for \( M \) in a similar manner.

4. A Brief Overview of the Algorithm

Since the truth of a CTL formula is conditional on the state of the input, the goal of our algorithm is to find all of the possible inputs which can satisfy the CTL formula at each state. This is done by labelling each state, \( s \), with a formula, \( \text{label}(s, f) \), from \( L(f) \) such that:

\[ M, s, A \models f \iff A \text{ sat } \text{label}(s, f). \]

The main procedure is shown in figure 4-1. Once the type of CTL formula is determined, \( \text{LabelGraph} \) is called recursively to determine the truth conditions for all of the subformulas of \( f \) (these conditions are needed in the labelling routines). Then, the labelling routine for the specific type of CTL formula is called to set \( \text{label}(s, f) \). After the algorithm terminates, the truth conditions for \( f \) and all of the subformulas of \( f \) are known.

```
procedure LabelGraph(f)
  if f = p then
    begin
      AtomicLabel(f);
      return;
    end;
  if f = \neg f then
    begin
      LabelGraph(f);
      NotLabel(f);
      return;
    end;
  ...;
end
```

Figure 4-1: The Main Procedure, LabelGraph
In order to handle fairness, we assume that \( \text{label}(s,c) \) and \( \text{label}(s,fair) \) are formulas from \( L(l) \) such that:

\[
\begin{align*}
A \text{ sat } \text{label}(s,c) & \iff M,s,A \models c_i \\
A \text{ sat } \text{label}(s,fair) & \iff \exists \text{ a fair path in } M' \text{ from } \langle s, A \rangle
\end{align*}
\]

5. Correctness of the Algorithm

After \( \text{LabelGraph}(l) \) is called, \( \text{label}(s,0) \) is set to a formula from \( L(l) \). If the algorithm is correct,

\[
M,s,A \models f \iff A \text{ sat } \text{label}(s,f)
\]

This will be proven by cases. Due space limitations, we will only give complete proofs for two cases: \( f \) is an atomic proposition and \( f \) is \( E[f_1 \cup f_2] \). (The proofs of the other procedures are similar. The other procedures are given without proof in Appendix II.)

5.1. \( f \) is an atomic proposition.

procedure AtomicLabel(f)

\[
\text{for all } s \in S \text{ do}
\]

\[
\text{if } f \notin l \text{ then}
\]

\[
\text{if } f \in P(s) \text{ then}
\]

\[
\text{label}(s,f) = \text{true};
\]

\[
\text{else}
\]

\[
\text{label}(s,f) = \text{false};
\]

\[
\text{else}
\]

\[
\text{label}(s,f) = t;
\]

end

Figure 5-1: The Procedure That Labels Atomic Propositions

By the definition of \( \models f \) for an atomic proposition,

\[
M,s,A \models f \iff M',\langle s, A \rangle \models f
\]

\[
\iff f \in P'(\langle s, A \rangle).
\]

Since \( P'(\langle s, A \rangle) = P(s) \cup A \),

\[
M,s,A \models f \iff f \in P(s) \lor f \in A.
\]

There are three cases that must be considered.

1. \( f \in l \land f \in P(s) \).

Since \( f \in l \land f \in A \), so

\[
M,s,A \models f \iff f \in P(s).
\]

Since \( f \in P(s) \) is our hypothesis, \( M,s,A \models f \) is true. But AtomicLabel sets \( \text{label}(s,f) \) to true, so \( A \text{ sat } \text{label}(s,f) \) is true for any \( A \). Therefore,

\[
M,s,A \models f \iff A \text{ sat } \text{label}(s,f).
\]
2. \( f \in f \land f \in P(s) \).

Since \( f \in f \land f \in A \), so

\[
M,s,A \vdash f \Leftrightarrow f \in P(s)
\]

Since \( f \in P(s) \) is our hypothesis, \( M,s,A \vdash f \) is false. But AtomicLabel sets label\((s,f)\) to false, so \( A \) sat label\((s,f)\) is false for any \( A \). Therefore,

\[
M,s,A \vdash f \Rightarrow A \text{ sat label}(s,f).
\]

3. \( f \in f \).

Since \( f \in f \land f \in P(s) \), so

\[
M,s,A \vdash f \Leftrightarrow f \in A
\]

In this case, AtomicLabel sets label\((s,f)\) to \( f \), so by the definition of sat, \( A \) sat label\((s,f)\) is \( f \in A \). Therefore,

\[
M,s,A \vdash f \Leftrightarrow A \text{ sat label}(s,f).
\]

Therefore, AtomicLabel is correct.

5.2. \( f = E[f_1 \cup f_2] \)

EUlabel uses two arrays of booleans, marked and stacked, that are indexed by the states of the machine. The intent is that marked\([s]\) should be true if there is a state \(<s,A>\) in \( M' \) that satisfies \( E[f_1 \cup f_2] \), and stacked\([s]\) should be true if a call eu\((s)\) is in progress.

Since LabelGraph calls itself recursively before EUlabel is called, we use recursive induction to establish:

\[
M,s,A \vdash f_1 \Leftrightarrow A \text{ sat label}(s,f_1), \text{ and} \quad M,s,A \vdash f_2 \Leftrightarrow A \text{ sat label}(s,f_2).
\]

Let \( INV \) be an abbreviation for the formula:

\[
\forall s [\text{marked}[s] \rightarrow
\exists A [M,s,A \vdash E[f_1 \cup f_2]] \&
\forall n \geq 0 \forall \text{paths in } M', \langle t_0,B_0,t_1,B_1,\ldots,t_n,B_n \rangle
\quad [s = t_n \& \forall 0 \leq i < n [M,t_i,B_i \vdash f_1] \rightarrow
\quad \forall 0 \leq i < n [\neg \text{marked}[t_i] \rightarrow \neg \text{marked}[t_i+1] \lor \text{stacked}[t_{i+1}]]] ] \&
\forall s [\text{stacked}[s] \rightarrow \text{marked}[s]].
\]

Lemma 1: The subroutine eu satisfies the assertion:

\[
\{ INV \& \exists A [M,s,A \vdash E[f_1 \cup f_2]] \& \neg \text{marked}[s] \& \neg \text{stacked}[s] \}
\quad \text{eu}(s)
\]

\[
\{ INV \& \forall s [\text{stacked}[s] = \text{stacked}[s'] \& \text{marked}[s]]
\]

where stacked\([s]'\) is the value of stacked\([s]\) before the call.
procedure EUlabel(f)
  for all $s \in S$ do
    begin
      marked[$s$] = $false$;
      stacked[$s$] = $false$;
    end;
  for all $s \in S$ do
    if $\neg$ marked[$s$] & (label($s$, $f_1$) $\land$ label($s$, $fair$)) $\neq$ $false$ then
      eu($s$);
  for all $s \in S$ do
    begin
      label($s$, $f_1$) = label($s$, $f_2$) $\land$ label($s$, $fair$);
      for all $(s,t,G) \in R$ do
        if marked[$t$] then
          label($s$, $f_1$) = ($G \land$ label($s$, $f_1$)) $\lor$ label($s$, $f_1$);
    end;
  end

procedure eu($s$)
  begin
    stacked[$s$] = $true$;
    marked[$s$] = $true$;
    for all $(t,s,G) \in R$ do
      if $\neg$ marked[$t$] & $\neg$ stacked[$t$] & (label($t$, $f_1$) $\land$ $G$) $\neq$ $false$ then
        eu($t$);
    stacked[$s$] = $false$;
  end

Figure 5-2: The Procedure That Labels E[f_1 U f_2]
∀ s [stacked[s] → marked[s]].

Now, since ¬marked[s] is part of the precondition, ¬marked[s] ∨ stacked[s] is true initially. After the first two statements are executed, stacked[s] is true, so ¬marked[s] ∨ stacked[s] is still true. Since nothing else has changed, the value of the implication

¬marked[t] → ¬marked[t] ∨ stacked[t]

hasn't changed. Therefore, the invariant still holds for all states other than s. Now that state s is marked, we must show that:

∃ A [M,s,A ⊨ E[t,U t]] &

∀ n ≥ 0 ∀ paths in M', (t_0,B_0,t_1,B_1, ..., t_n,B_n)

[s = t_n & ∀ 0 ≤ i < n [t_i,B_i ⊨ f_i] & ∀ 0 ≤ i < n [¬marked[t_i] → ¬marked[t_i+1] ∨ stacked[t_i+1]]]

The existence of A is given by the precondition. We can prove the second part by contradiction. Assume that

∀ n ≥ 0 ∃ path in M', (t_0,B_0,t_1,B_1, ..., t_n,B_n)

[s = t_n & ∀ 0 ≤ i < n [t_i,B_i ⊨ f_i] & ∃ 0 ≤ i < n [¬marked[t_i] & ¬marked[t_i+1] & ¬stacked[t_i+1]]]

Since stacked[s] is true but stacked[t_i+1] is false, we conclude that t_i+1 ≠ s. But since t_i+1 ≠ s, we know the invariant holds. Since marked[t_i+1] and (t_i,B_i) ⊨ f_i, we have:

⟨t_i,B_i⟩ ⊨ f_i → ¬marked[t_i+1] ∨ stacked[t_i+1].

Since we have assumed ⟨t_i,B_i⟩ ⊨ f_i and ¬marked[t_i], we conclude ¬marked[t_i+1] ∨ stacked[t_i+1]. This contradicts marked[t_i+1] & ¬stacked[t_i+1], so the invariant is true for state s. Therefore, the first two statements preserve INV.

Now we want to show that the loop preserves INV. By assumption, a recursive call preserves INV if the precondition is satisfied before the call. INV is true before the loop. The recursive call eu(t) is made only if (t,s,G) ∈ / & ¬marked[t] & ¬stacked[t] & (label(t) ∧ G) ≠ false, so ¬marked[t] & ¬stacked[t] is true before the call. Now it is necessary to show that:

∃ A [M,t,A ⊨ E[t,U t]]

Now,

(label(t) ∧ G) ≠ false ⇒ ∃ A [A sat label(t) ∧ G]

⇒ ∃ A [A sat label(t) ∧ A sat G].

By the definition of R' and the inductive hypothesis,

(t,s,G) ∈ R & A sat G → ∀ B [(⟨t,A⟩,⟨s,B⟩) ∈ R'], so

∃ A [M,t,A ⊨ f_i & ∀ B [(⟨t,A⟩,⟨s,B⟩) ∈ R']].

By the preconditions, ∃ B [M,s,B ⊨ E[t,U t]], so by definition of ⊨ and M':
\exists \text{ fair path } \langle u_0, C, u_1, C, \ldots \rangle \exists n \geq 0
\text{[s = u_0 \& M, u_n, C_n \models f_1 \& \forall 0 \leq i < n [M, u_i, C_i \models f_i]}.\]

Since \( M, t, A \models f_1 \) and \( \langle t, A, \langle s, B \rangle \rangle \in R' \), we can add \( \langle t, A \rangle \) to the beginning of this path. This path must also be fair, so we have:
\[
\exists \text{ fair path } \langle u_0, C_0, u_1, C_1, \ldots \rangle \exists n \geq 0
\text{[t = u_0 \& M, u_n, C_n \models f_1 \& \forall 0 \leq i < n [M, u_i, C_i \models f_i]}.\]

Therefore, \( \exists A [M, t, A \models E[f_1 \cup f_2]] \), so the precondition is satisfied when the loop is entered. After a recursive call, \( INV \) is preserved, so the precondition is satisfied for all recursive calls. Therefore, \( INV \) is true after leaving the loop.

Finally, we must show that the final statement preserves \( INV \). To begin with, the implication \( \text{stacked}[s] \rightarrow \text{marked}[s] \) is true after the final statement, since \( \text{stacked}[s] \) is false. Since nothing else has changed, we have
\[
\forall s \ [\text{stacked}[s] \rightarrow \text{marked}[s]].
\]

Since the loop makes a recursive call only if \( (t, s, G) \in R \& \neg \text{marked}[t] \& \neg \text{stacked}[t] \) \& (\text{label}(t, f_1) \land G) \neq \text{false}, \) and \( \text{marked}[t] \) is part of the postcondition, we can conclude:
\[
\forall t \ [\neg ((t, s, G) \in R \& \neg \text{marked}[t] \& \neg \text{stacked}[t] \& (\text{label}(t, f_1) \land G) \neq \text{false})]
\]

Using the same argument as above,
\[
(t, s, G) \in R \& (\text{label}(t, f_1) \land G) \neq \text{false} \Rightarrow \exists A [M, t, A \models f_1 \& \forall B [\langle t, A, \langle s, B \rangle \rangle \in R']],\]
so
\[
\forall t \ [\exists A [M, t, A \models f_1 \& \forall B [\langle t, A, \langle s, B \rangle \rangle \in R'] \rightarrow \text{marked}[t] \lor \text{stacked}[t]].
\]

By the definition of \( R' \),
\[
\exists B [\langle t, A, \langle s, B \rangle \rangle \in R'] = (t, s, G) \in R \& A \text{ sat } G
= \forall B [\langle t, A, \langle s, B \rangle \rangle \in R'],\]
so
\[
\forall t \ [\exists A [M, t, A \models f_1 \& \exists B [\langle t, A, \langle s, B \rangle \rangle \in R'] \rightarrow \text{marked}[t] \lor \text{stacked}[t]].
\]

We will now show that \( INV \) is true by contradiction. Setting \( \text{stacked}[s] \) to false cannot change the fact that
\[
\forall t \ [\text{marked}[t] \rightarrow \exists A [M, s, A \models E[f_1 \cup f_2]]].
\]

Therefore, if \( INV \) is false, we must have:
\[
\exists t \ [\text{marked}[t] \& \exists n \geq 0 \exists \text{ path in } M', \langle u_0, C_0, u_1, C_1, \ldots, u_n, C_n \rangle \]
[\[t = u_n \& \forall 0 \leq i < n [M, u_i, C_i \models f_i] \& \exists 0 \leq i < n [\neg \text{marked}[u_i] \& \text{marked}[u_{i+1}] \& \neg \text{stacked}[u_{i+1}]]].\]

Since the only change is that \( \text{stacked}[s] \) is false and \( INV \) was true initially, this can only be true if \( u_{i+1} = s \). Now, we have assumed \( M, u_i, C_i \models f_1 \) and \( \langle u_r, C_r, u_{i+1}, C_{i+1} \rangle \in R' \), so from the loop's postcondition we can conclude \( \text{marked}[u_i] \lor \text{stacked}[u_i] \). But since \( \forall s \ [\text{stacked}[s] \rightarrow \text{marked}[s]] \), we conclude \( \text{marked}[u_i] \). But this contradicts \( \neg \text{marked}[u_i] \), so \( INV \) must be preserved by the final statement.
Therefore, the lemma is true. □

Lemma 2: After the loop in EUlabel that calls eu has terminated,
\[ \forall s \, \text{marked}[s] \Rightarrow \exists A \, [M,s,A \models E[f_1,U,f_2]]. \]

Proof: To begin with, we want to show that whenever eu is called, the precondition is satisfied. Initially all states are unmarked and unstacked, so \( \text{INV} \land \neg \text{stacked}[s] \) is trivially true. Since eu is only called if \( \neg \text{marked}[s] \land (\text{label}(s,f_2) \land \text{label}(s,\text{fair})) \land \text{false} \) is true, \( \neg \text{marked}[s] \) is true. By the inductive hypothesis and definition of \( \text{label}(s,\text{fair}) \),
\[
\text{label}(s,f_2) \land \text{label}(s,\text{fair}) \land \text{false} = \exists A[Asat\text{label}(s,f_2) \land \text{label}(s,\text{fair})]
\]
\[
= \exists A[A \text{sat} \text{label}(s,f_2) \land A \text{sat} \text{label}(s,\text{fair})]
\]
\[
= \exists A[M,s,A \models f_2 \land \exists a \text{ fair path from } <s,A>].
\]
So by definition, \( \exists A[M,s,A \models E[f_1,U,f_2]] \) is true. Therefore, the precondition is true when the loop is entered. Since eu preserves \( \text{INV} \) and doesn't change the array \( \text{stacked} \), the precondition is true for all iterations of the loop.

After the loop has terminated, \( \text{INV} \) is still true. Therefore, the forward implication of this lemma is true. In order to prove the reverse implication, we start by showing that \( \text{INV} \) implies:
\[ \forall s \, [\neg \text{marked}[s] \rightarrow \forall n \geq 0 \, \forall \text{ paths in } M', <t_0,B_0>, <t_1,B_1>, ..., <t_n,B_n>
\]
\[ [s = t_0 \land \forall 0 \leq i < n [M,t_i,B_i \models f_1] \land M,t_n,B_n \models f_2 \rightarrow \neg \text{marked}[t_n]]. \]
Assume that this is false, so we have:
\[ \exists s \, [\neg \text{marked}[s] \land \exists n \geq 0 \exists \text{ path in } M', <t_0,B_0>, <t_1,B_1>, ..., <t_n,B_n>
\]
\[ [s = t_0 \land \forall 0 \leq i < n [M,t_i,B_i \models f_1] \land M,t_n,B_n \models f_2 \land \text{marked}[t_n]]. \]
Since \( t_n \) is marked, \( \text{INV} \) tells us that
\[ \forall 0 \leq i < n [\neg \text{marked}[t_i] \rightarrow \neg \text{marked}[t_{i+1}] \lor \text{stacked}[t_{i+1}]]. \]
Since the array \( \text{stacked} \) is initialized to be false and eu doesn't change the array, \( \text{stacked}[t_{i+1}] \) is false, so
\[ \forall 0 \leq i < n [\neg \text{marked}[t_i] \rightarrow \neg \text{marked}[t_{i+1}]], \text{or equivalently}
\]
\[ \forall 0 < i \leq n [\text{marked}[t_i] \rightarrow \text{marked}[t_{i+1}]]. \]
Since we know \( \text{marked}[t_n] \), we can apply modus ponens \( n \) times and conclude \( \text{marked}[t_0] \). But since \( s = t_0 \), this contradicts our hypothesis \( \neg \text{marked}[s] \). Therefore, the statement is true.

Since eu always marks its argument and stacked is initialized to be false, the postcondition for the entire loop is:
\[ \text{INV} \land \forall s \, [\neg \text{stacked}[s]] \land \forall s \, [\neg (\neg \text{marked}[s] \land \text{label}(s,f_2) \land \text{label}(s,\text{fair}) \land \text{false})], \text{or} \]
INV & $\forall s \left[ -\text{marked}[s] \right] \land \forall s \left[ \text{label}(s,f_2) \land \text{label}(s,fair) \land \text{false} \rightarrow \text{marked}[s] \right]$

By the inductive hypothesis and the definition of label(s,fair),

\[
\text{label}(s,f_2) \land \text{label}(s,fair) \land \text{false} \implies \exists A \left[ \text{A sat label}(s,f_2) \land \text{A sat label}(s,fair) \right]
\]

\[
\implies \exists A \left[ M, s, A \models f_2 \land \exists \text{a fair path from } <s,A> \right].
\]

So the loop's postcondition is:

\[
\text{INV} \land \forall s \left[ -\text{marked}[s] \right] \land \forall s \left[ \exists A \left[ M, s, A \models f_2 \land \exists \text{a fair path in } M' \text{ from } <s,A> \right] \rightarrow \text{marked}[s] \right].
\]

We have shown that if a state is unmarked and there is a path from this state that satisfies $E[f_1 f_2]$, then the state on this path that satisfies $f_2$ must be unmarked. But since all states satisfying $f_2$ on a fair path have been marked when the loop is exited, we conclude that if there is a path from an unmarked state that satisfies $E[f_1 f_2]$, the path must be unfair. Therefore,

\[
\forall s \left[ -\text{marked}[s] \rightarrow \neg \exists A \left[ M, s, A \models E[f_1 f_2] \right] \right].
\]

This is the reverse implication, so the lemma is proven. □

The second loop sets label(s,f) such that:

\[
\text{label}(s,f) = \text{label}(s,f_2) \land \text{label}(s,fair) \lor \bigvee \left\{ G \land \text{label}(s,f) \left[ (s,t,G) \in R \land \text{marked}[r] \right] \right\}, \text{ so } \text{A sat label}(s,f) \iff \text{A sat label}(s,f_2) \land \text{label}(s,fair) \lor \exists t \left[ (s,t,G) \in R \land \text{marked}[r] \land \text{A sat G} \land \text{A sat label}(s,f_2) \right].
\]

Using the inductive hypothesis and the definitions of $R'$ and label(s,fair), we get

\[
\text{A sat label}(s,f) = \left( M, s, A \models f_2 \land \exists \text{a fair path in } M' \text{ from } <s,A> \right) \lor \exists t \left[ \text{marked}[r] \land \forall B \left[ (<s,A>,<t,B>) \in R' \right] \land M, s, A \models f_1 \right].
\]

Since marked[1] \iff \exists B \left[ M, t, B \models E[f_1 f_2] \right], this becomes:

\[
\text{A sat label}(s,f) = \left( M, s, A \models f_2 \land \exists \text{a fair path in } M' \text{ from } <s,A> \right) \lor \exists t \left[ \exists B \left[ M, t, B \models E[f_1 f_2] \right] \land \forall B \left[ (<s,A>,<t,B>) \in R' \right] \land M, s, A \models f_1 \right].
\]

This implies:

\[
\text{A sat label}(s,f) = \left( M, s, A \models f_2 \land \exists \text{a fair path in } M' \text{ from } <s,A> \right) \lor \exists t \left[ \exists B \left[ M, t, B \models E[f_1 f_2] \right] \land (M',<s,A>) \models f_1 \land \text{EX E}[f_1 f_2] \right].
\]

Using the definition of $\models$ for $M$ and regrouping:

\[
\text{A sat label}(s,f) = \left( M',<s,A> \models f_2 \land \exists \text{a fair path from } <s,A> \right) \lor \left( M',<s,A> \models f_1 \land \text{EX E}[f_1 f_2] \right).
\]

From $M,t,B \models E[f_1 f_2]$, we can infer that $\exists \text{a fair path in } M' \text{ from } <t,B>$. Therefore, using the definitions of EX and $\land$, we get:

\[
\text{A sat label}(s,f) = \left( M',<s,A> \models f_2 \land \exists \text{a fair path from } <s,A> \right) \lor \left( M',<s,A> \models f_1 \land \text{EX E}[f_1 f_2] \right).
\]

Since this is the fixpoint characterization of $E[f_1 f_2]$ for fair paths, we have:

\[
\text{A sat label}(s,f) = \left( M',<s,A> \models E[f_1 f_2] \right)
\]
\[ M,s,A \models E[1, U 2]. \]

Therefore, EUlabel is correct.

6. Complexity of the Model Checker

In this section, we consider the complexity of verifying CTL formula \( f \) in structure \( M = (S, l, O, R, P) \). We assume that the conditions in \( R \) are given in disjunctive normal form. Furthermore, let \( |f| \) be the length of \( f \), \( |l| \) be the number of input propositions, and \( |R| \) be the total number of conjuncts in the transition conditions.

We should begin by pointing out that the new algorithm is no worse than the original algorithm.

**Theorem 3:** The complexity of verifying CTL formula \( f \) in structure \( (S, l, O, R, P) \) is \( O(|f| \cdot 2^{|l|} \cdot |R|) \).

**Proof:** To begin with, we will represent all of the label\((s, f)\) formulas by an array of \( 2^{|l|} \) booleans, one for each possible input assignment. If an array element is true, then the corresponding input assignment satisfies the formula that we are representing. It is easy to see that negation, disjunction, and conjunction of formulas in this form requires \( O(2^{|l|}) \) time (we only need to examine each array element once). It is also possible to keep track of whether there are any elements set to true, so satisfiability can be done in constant time.

Now, it can be shown that each labelling procedure does a constant amount of boolean formula manipulation and other work for each state and each transition. Therefore, the complexity of each procedure is \( O(2^{|l|} \cdot (|S| + |R|)) \). But since each state has at least one transition, \( |S| \) is \( O(|R|) \), so the complexity of each procedure is \( O(2^{|l|} \cdot |R|) \). Since a labelling procedure is called once for each operator in \( f \), it is clear that \(|f| \) calls are made. Therefore, the complexity of the algorithm is \( O(|l| \cdot 2^{|l|} \cdot |R|) \). \( \square \)

It is doubtful if a different representation for the boolean formulas would give a better result. After all, if the formula \( f \) contains only inputs and propositional operators, the model checking problem reduces to boolean satisfiability, which is NP-hard. (As it is, we avoid being exponential in the length of the formula by being exponential in the number of inputs instead.)

Although in general the new algorithm has the same complexity as the original, if we restrict the logic, we can get a much better result. Consider the following definition of allowable CTL formulas:

1. \( \text{true} \) is an allowable CTL formula.
2. If \( p \) is an atomic proposition and \( f \) is an allowable CTL formula, then \( p \land f \) and \( \neg p \land f \) are allowable.
3. If $f_1$ and $f_2$ are allowable CTL formulas, then so are $f_1 \lor f_2$, $\exists X f_1$, $E[G f_1]$, and $E[f_1 U f_2]$.

In practice, we have found that this class of formulas is sufficiently general for most purposes. Even though no universally quantified formulas are allowable, it is usually the case that the negation of such a formula is allowable.

This definition of allowable was chosen because it allows us to specify an upper bound on the complexity of all of the label$(s,f)$ formulas. This bound is given in the following lemma, which will not be proven here:

**Lemma 4:** If $f$ is an allowable CTL formula, then all of the formulas label$(s,f)$ can be represented in a data structure that takes up $O(|l| \cdot |R| \cdot |f|)$ space. Moreover, determining if (label$(s,f)$ $\land$ label$(s,fa/r)$ $\land$ false) can be done in constant time and determining if (label$(s,f) \land G$ $\land$ false) (where $(s,t,G) \in R$) can be done in $O(|G|)$ time (where $|G|$ is the number of conjuncts in $G$).

The idea is that label$(s,f)$ can be written as

$$\bigvee_i B_i \land \bigvee_j (C_{ij} \land G_j)$$

where $B_i$ is a conjunct of input literals, $C_{ij}$ is either true or false, and $G_j$ is a conjunct in a transition condition from state $s$. Since it can be shown that $i$ is $O(|l|)$ and the total of the $j$ for all states is $O(|R|)$, we can represent all of the label$(s,f)$ as an array of $|R|$ rows and $O(|l|)$ columns. Each element in the array is a $B_i \land G_j \land C_{ij}$ conjunct that can be represented by an array of size $|l|$. Therefore, the total amount of space is $O(|l| \cdot |R| \cdot |f|)$. In addition, if we keep track of how many columns are not empty in each row, we can determine if (label$(s,f) \land G$ $\land$ false) in $O(|G|)$ time by testing if any of the $|G|$ rows corresponding to a conjunct in $G$ have a column with a non-empty value. Furthermore, we can keep track of whether any row corresponding to a fair transition has a non-empty column, so that we can determine if (label$(s,f) \land label$(s,fa/r)$ $\land$ false) in constant time.

As a consequence of this lemma, we can show that:

**Theorem 5:** The complexity of verifying allowable CTL formula $f$ in structure $(S, l, O, R, P)$ is $O(|f|^2 \cdot |l| \cdot |R|)$.

**Proof:** As before, there are $O(|l|)$ calls made to labelling routines. But because of the above lemma, it can be shown that the worst case for a single labelling routine is $E[G f_1]$, which takes $O(|l| \cdot |l| \cdot |R| \cdot |c_j|)$ time. (The dominant term is due to the loop that must find if (label$(s,f_1) \land label$(s,c_j) \land G$ $\land$ false) for each fairness constraint $c_j$. The complexity of finding the conjunction a disjunction of $O(|l|)$ conjuncts and a disjunction of $O(|c_j|)$ conjuncts is $O(|l| \cdot |c_j|)$ times the complexity of finding the conjunction of two conjuncts (in this case, $O(|l|)$). Summing over all fairness constraints and all transitions
gives the above result.) If we treat the number and size of the fairness constraints as a constant, the complexity of $EG_f$ is $O(|f|\cdot|R|)$. But since $|f|_f$ is $O(|f|)$ and at most $|f|$ calls to labelling routines are made, the algorithm's complexity is $O(|f|^2\cdot|f|\cdot|R|)$. □

If we have more information about the structure that we are verifying, we can add more formulas into our "allowable" class. For example, a structure $(S, I, O, R, P)$ is deterministic iff for all $(s,t,G)$ and $(s,u,H)$ in $R$, either $t = u$ or $G \land H$ is unsatisfiable. (i.e. Given an input assignment, there is exactly one possible next state.) Then, for a deterministic structure, we have the following results:

Lemma 6: For any CTL formula, $f$, label$(s, \Box f)$ can be written as

$$\forall f (C_f \land G_f)$$

where $C_f$ is either true or false, and $G_f$ is a conjunct in a transition condition from state $s$.

This is obvious from the labelling routine for $\Box f$:

```
procedure $\Box$label($f$)
  for all $s \in S$ do
    begin
      label$(s, f) = false$;
      for all $(s,t,G) \in R$ do
        if label$(t,f) \land label(t, fair) = false$ then
          label$(s,f) = G \lor label(s,f)$;
        end;
    end;
end
```

Lemma 7: For any CTL formula, $f$, label$(s, \neg \Box f)$ can be written as

$$\forall f (C_f \land G_f)$$

where $C_f$ is either true or false, and $G_f$ is a conjunct in a transition condition from state $s$.

Since all of the $G_f$ are disjoint and include all possible input combinations, we can negate the formula in the previous lemma by simply negating all of the $C_f$!

Theorem 8: If the formula $\neg \Box f$ is added to the definition of allowable, then lemma 4 and theorem 5 are still true.

From the lemma above, it is clear that label$(s,\neg \Box f)$ takes $O(|f| \cdot |R|)$ space, which is clearly also $O(|f|\cdot|R|\cdot|R|)$. We can easily keep track of the same information as before, so lemma 4 still holds. Furthermore, the previous lemma also gives an algorithm for negating $\Box f$ that takes $O(|R|)$ time, so $EG f$ is still the worst case in the proof of theorem 5. Therefore, the proof still holds, so theorem 5 is true for the extended definition of "allowable". □
7. Conclusion

In this paper, we have presented an enhancement to the CTL model checking algorithm that is capable of dealing with conditional transitions. Although in the worst case, the complexity is the same as the original algorithm ($O(|F|^2|R|)$), the new algorithm performs much better than the original in practice. The difference in performance can be explained by the fact that most of the formulas that we have attempted to verify in practice belong to a special class for which the complexity of the new algorithm is $O(|F|^2|R|)$. 

As it stands, our new algorithm can be used to verify Moore machines directly. In the near future, we plan to enhance it to deal with Mealy machines as well. (We can currently simulate Mealy outputs by treating them as inputs and adding a special error state that the machine enters and remains in whenever the "outputs" are incorrect. Then, we can add a fairness constraint that states that we are not in the error state infinitely often. Therefore, any transition that enters the error state is unfair, so these transitions are never considered and the "outputs" are always correct.)

I. Computing the Fairness Labels

In LabelGraph, we assumed that we had access to $\text{label}(s,c_f)$ and $\text{label}(s,\text{fair})$. In this section, we describe how these labels are calculated.

If there is a fair path in $M'$ from $<s,A>$ is always true, it is easy to see that $M,s,A \models f \Rightarrow M,s,A \models f$ for any CTL formula $f$. Therefore, if $\text{label}(s,\text{fair}) = \text{true}$ and the set of fairness constraints is empty, the preconditions of LabelGraph will be satisfied and the postcondition will be:

$$\text{A sat } \text{label}(s,f) \Rightarrow M,s,A \models f$$

$$\text{A sat } \text{label}(s,c_r) \Rightarrow M,s,A \models c_r$$

Therefore, we can use LabelGraph to check all of the $c_r$, so we will have:

$$\text{A sat } \text{label}(s,c_f) \Rightarrow M,s,A \models c_f$$

In order to set $\text{label}(s,\text{fair})$, we use the procedure SetFair. This procedure uses two arrays of booleans, marked and stacked, that are indexed by the states of the machine. The intent is that marked$[s]$ should be true if there is a fair path from some state $<s,A>$ in $M'$, and stacked$[s]$ should be true if a call fair$(s)$ is in progress. We assume that label$(s,c_f)$ is defined so that:

$$\text{A sat } \text{label}(s,c_f) \Rightarrow <s,A> \models c_f$$

procedure SetFair()
for all $s \in S$ do
begin
marked$[s] = \text{false}$;
end

stacked[s] = false;
end;
V = S;
E = \{(s,t) \mid s,t \in V \land (s,t,G) \in R\};
Find the strongly connected components of directed graph \((V, E)\);
for all \(V', a\) non-trivial strongly connected component of \((V, E)\) do
if \(\forall c, \exists r, t \in V' \ searchable[(r,t,G) \in R \land (\text{label}(r,c) \land G) = false]\) then
for all \(v \in V'\) do
if \(\neg \text{marked}(v)\) then
fair(v);
for all \(s \in S\) do
begin
\text{label}(s, fair) = false;
for all \((s,t,G) \in R\) do
if \text{marked}(t) then
\text{label}(s,t) = G \lor \text{label}(s,t)
end;
end
procedure fair(s)
begin
stacked[s] = true;
marked[s] = true;
for all \((t,s,G) \in R\) do
if \(\neg \text{marked}(t) \land \neg \text{stacked}(t)\) then
eg(t);
stacked[s] = false;
end

II. The Complete Algorithm

II.1. \(f = \neg f_1\)

\text{procedure NotLabel(f)}
\begin{algorithmic}
\FORALL{s \in S}
\STATE\text{label}(s, f) = \neg \text{label}(s, f_1);
\ENDFOR
\end{algorithmic}

II.2. \(f = f_1 \land f_2\)

\text{procedure AndLabel(f)}
\begin{algorithmic}
\FORALL{s \in S}
\STATE\text{label}(s, f) = \text{label}(s, f_1) \land \text{label}(s, f_2);
\ENDFOR
\end{algorithmic}
11.3. \( f = \textbf{EX} f_1 \)

procedure \textbf{EX}Label\( (f) \)

for all \( s \in S \) do

begin

label\( (s,f) = false; \)

for all \( (s,t,G) \in R \) do

if label\( (t,f_1) \land \text{label}(t,\text{fair}) = false \) then

label\( (s,f) = G \lor \text{label}(s,f); \)

end;

end

11.4. \( f = \textbf{AX} f_1 \)

Since \( \textbf{AX} f = \neg \textbf{EX} (\neg f) \), we can use NotLabel and \textbf{EX}Label to check \( \textbf{AX} f_1 \).

11.5. \( f = \textbf{A}[f_1 \cup f_2] \)

Since \( \textbf{A}[f_1 \cup f_2] = \neg (\textbf{E}[\neg f_1 \cup \neg f_2] \lor \textbf{E} f_2) \), we will give an algorithm for checking \( \textbf{EG} f \) and use this routine, NotLabel, EUlabel, and AndLabel to check this case.

11.6. \( f = \textbf{EG} f_1 \)

This procedure uses two arrays of booleans, marked and stacked, that are indexed by the states of the machine. The intent is that marked\( [s] \) should be true if there is a state \( \langle s,A \rangle \) in \( M' \) that satisfies \( \textbf{EG} f_1 \), and stacked\( [s] \) should be true if a call \( \textbf{eg}(s) \) is in progress.

procedure \textbf{EG}Label\( (f) \)

for all \( s \in S \) do

begin

marked\( [s] = false; \)

stacked\( [s] = false; \)

end;

\( V = \{ s \mid \text{label}(s,f_1) = false \}; \)

\( E = \{ (s,t) \mid s,t \in V \land (s,t,G) \in R \land \text{label}(s,f_1) \land G = false \}; \)

Find the strongly connected components of directed graph \( (V,E) \);

for all \( V' \), a non-trivial strongly connected component of \( (V,E) \) do

if \( \forall c, \exists r,t \in V' \langle (r,t,G) \in R \land \text{label}(r,f_1) \land \text{label}(r,c) \land G = false \rangle \) then

for all \( v \in V' \) do

if \( \neg \text{marked}[v] \) then

\( \text{eg}(v); \)

for all \( s \in S \) do

begin

label\( (s,f) = false; \)

for all \( (s,t,G) \in R \) do

if \( \text{marked}[r] \) then

label\( (s,f) = G \land \text{label}(s,f_1) \lor \text{label}(s,f) \)

end;

end
procedure eg(s)
begin
  stacked[s] = true;
  marked[s] = true;
  for all (i,s,G) ∈ R do
    if ¬marked[i] & ¬stacked[i] & (label(i,f) ∧ G) = false then
      eg(i);
      stacked[s] = false;
  end
References


Table of Contents

1. Introduction
2. The Specification Language
3. The Finite State System Description
4. A Brief Overview of the Algorithm
5. Correctness of the Algorithm
   5.1. \( f \) is an atomic proposition.
   5.2. \( f = E[f_1 \cup f_2] \)
6. Complexity of the Model Checker
7. Conclusion
   I. Computing the Fairness Labels
   II. The Complete Algorithm
      II.1. \( f = \neg f_1 \)
      II.2. \( f = f_1 \land f_2 \)
      II.3. \( f = \exists x f_1 \)
      II.4. \( f = A x f_1 \)
      II.5. \( f = A[f_1 \cup f_2] \)
      II.6. \( f = EG f_1 \)
List of Figures
Figure 4-1: The Main Procedure, Labelgraph 4
Figure 5-1: The Procedure That Labels Atomic Propositions 5
Figure 5-2: The Procedure That Labels $E[f_1 U f_2]$ 7