DESIGN AND SYNTHESIS OF SYNCHRONIZATION SKELETONS

USING BRANCHING TIME TEMPORAL LOGIC

Edmund M. Clarke

E. Allen Emerson
Aiken Computation Laboratory
Harvard University
Cambridge, Mass. 02138

This work was partially supported by NSF Grant MCS-7908365.
ABSTRACT

We present a method of constructing concurrent programs in which the synchronization skeleton of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. Because the synchronization skeleton is in general finite state, its properties can be specified by a formula \( f \) in a propositional Temporal Logic. (The synthesis method uses a decision procedure based on the finite model property of the logic to determine satisfiability of \( f \).) If the formula \( f \) is satisfiable, then the specification it expresses is consistent, and a model for \( f \) with a finite number of states is constructed. The synchronization skeleton of a program meeting the specification can be read from this model. If \( f \) is unsatisfiable, the specification is inconsistent.

In the traditional approach to concurrent program verification, the proof that a program meets its specification is constructed using various axioms and rules of inference in a deductive system such as temporal logic. The task of proof construction can be quite tedious, and a good deal of ingenuity may be required. We believe that this task may be unnecessary in the case of finite state concurrent systems, and can be replaced by a mechanical check that the system meets a specification expressed in a propositional temporal logic. The global system flowgraph of a finite state concurrent system may be viewed as defining a finite structure. We describe an efficient algorithm (a model checker) to decide whether a given finite structure is a model of a particular formula. We also discuss extended logics for which it is not possible to construct efficient model checkers.
1. INTRODUCTION

We propose a method of constructing concurrent programs in which the synchronization skeleton of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic can be used to specify their properties.

Our synthesis method exploits the (bounded) finite model property for an appropriate propositional Temporal Logic which asserts that if a formula of the logic is satisfiable, it is satisfiable in a finite model (of size bounded by a function of the length of the formula). Decision procedures have been devised which, given a formula of Temporal Logic, \( f \), will decide whether \( f \) is satisfiable or unsatisfiable. If \( f \) is satisfiable, a finite model of \( f \) is constructed. In our application, unsatisfiability of \( f \) means that the specification is inconsistent (and must be reformulated). If the formula \( f \) is satisfiable, then the specification it expresses is consistent. A model for \( f \) with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The finite model property ensures that any program whose synchronization properties can be expressed in propositional Temporal Logic can be realized by a system of concurrently running processes, each of which is a finite state machine.
Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment and for monitors. However, we believe that it is also possible to extend these techniques to synthesize distributed programs. One such application would be the automatic synthesis of network communication protocols from propositional Temporal Logic specifications.

Previous efforts toward parallel program synthesis can be found in the work of [LA78] and [RK80]. [LA78] uses a specification language that is essentially predicate calculus augmented with a special predicate to define the relative order of events in time. [RK80] uses an applied linear time Temporal Logic. Both [LA80] and [RK80] use ad hoc techniques to construct a monitor that meets the specification. We have recently learned that [WO81] has independently developed model-theoretic synthesis techniques similar to our own. However, he uses a linear time logic for specification and generates CPS-like programs.

We also discuss how a Model Checker for Temporal Logic formulae can be used to verify the correctness of a priori existing programs. In the traditional approach to concurrent program verification, the proof that a program meets its specification is constructed using various axioms and rules of inference in a deductive system such as temporal logic. The task of proof construction can be quite tedious, and a good deal of ingenuity may be required. We believe that this task may be unnecessary in the case of finite state concurrent systems, and can be replaced by a mechanical check that the system meets a specification expressed in a propositional temporal logic. The global system flowgraph of a finite state concurrent system may be viewed as defining a finite structure. We describe an efficient algorithm (a model checker) to decide whether a given finite structure is a model of a particular formula. We also discuss extended logics for which it is not possible to construct efficient model checkers.
The paper is organized as follows: Section 2 discusses the model of parallel computation. Section 3 presents the branching time logic that is used to specify synchronization skeletons. Fixpoint characterizations for various temporal operators are given in Section 4. Sections 5 and 6 describe the model checker and the decision procedure, respectively. Finally, Section 7 shows how the synthesis method can be used to construct a solution to the starvation free mutual exclusion problem.
2. MODEL OF PARALLEL COMPUTATION

We discuss concurrent systems consisting of a finite number of fixed processes $P_1, \ldots, P_m$ running in parallel. The treatment of parallelism is the usual one: nondeterministic interleaving of the sequential "atomic" actions of the individual processes $P_i$. Each time an atomic action is executed, the system "execution" state is updated. This state may be thought of as containing the location counters and the data values for all processes. The behavior of a system starting in a particular state may be described by a computation tree. Each node of the tree is labelled with the state it represents, and each arc out of a node is labelled with a process index indicating which nondeterministic choice is made, i.e., which process's atomic action is executed next. The root is labelled with the start state. Thus, a path from the root through the tree represents a possible computation sequence of the system beginning in a given start state. Our temporal logic specifications may then be thought of as making statements about patterns of behavior in the computation trees.

Each Process $P_i$ is represented as a flowgraph. Each node represents a region or a block of code and is identified by a unique label. For example there may be a node labelled $CS_i$ representing "the critical section of code of process $P_i".$ Such a region of code is uninterpreted in that its internal structure and intended application are unspecified. While in $CS_i$, the process $P_i$ may simply increment variable $x$ or it may perform an extensive series of updates on a large database. The underlying semantics of the computation performed in the various code regions are irrelevant to the synchronization skeleton. The arcs between nodes represent possible transitions between code regions. The labels on the arcs indicate under
what conditions $P_i$ can make a transition to a neighboring node. Our job is to supply the enabling conditions on the arcs so that the global system of processes $P_1, \ldots, P_k$ meets a given Temporal Logic specification.
3. THE SPECIFICATION LANGUAGE

Our specification language is a (propositional) branching time temporal logic called Computation Tree Logic (CTL) and is based on the language presented in [EC80]. Our current notation is inspired by the language of "Unified Branching Time" (UB) discussed in [BMB1]. UB is roughly equivalent to that subset of the language presented in [EC80] obtained by deleting the infinitary quantifiers and the arc conditions and adding an explicit next-time operator. For example, in [EC80] we write \( \forall \text{path} \exists \text{node} P \) to express the inevitability of predicate \( P \). The corresponding formula in our UB-like notation is AFP. The language presented in [EC80] is more expressive than UB as evidenced by the formula \( \forall \text{path} \neg \forall \text{node} P \) (which is not equivalent to any formula in UB or in the language of [EC80] without infinitary quantifiers). However, the UB-like notation is more concise and is sufficiently expressive for the purposes of program synthesis.

We use the following syntax (where \( p \) denotes an atomic proposition and \( f_1 \) denotes a (sub-) formula):

1. Each of \( p, f_1 \land f_2, \) and \( \neg f_1 \) is a formula (where the latter two constructs indicate conjunction and negation, respectively).
2. \( \text{EX}_j f_1 \) is a formula which intuitively means that there is an immediate successor state reachable by executing one step of process \( p_j \) in which formula \( f_1 \) holds.
3. \( \text{A}[f_1 U f_2] \) is a formula which intuitively means that for every 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.
4. \( \text{E}[f_1 U f_2] \) is a formula which intuitively means that for some computation path, there exists an initial prefix of the path such
A structure

\begin{figure}
\begin{center}
\begin{tikzpicture}[node distance=2cm, thick, main/.style = {draw, circle}]
    \node [main, label=above:{\(S_0\)}] (a) {a \ b \ c} ;
    \node [main, below of=a, label=above:{\(S_1\)}] (b) {b} ;
    \node [main, right of=b, label=above:{\(S_2\)}] (c) {a \ c} ;

    \draw [->] (a) -- node[below] {1} (b) ;
    \draw [->] (b) -- node[above] {2} (c) ;
    \draw [->] (a) -- node[above] {2} (c) ;
\end{tikzpicture}
\end{center}
\end{figure}

The corresponding tree for start state \(S_0\)

\begin{figure}
\begin{center}
\begin{tikzpicture}[node distance=2cm, thick, main/.style = {draw, circle}]
    \node [main, label=above:{\(S_0\)}] (a) {} ;
    \node [main, below of=a, label=above:{\(S_1\)}] (b) {} ;
    \node [main, right of=b, label=above:{\(S_2\)}] (c) {} ;

    \draw [->] (a) -- node[below] {1} (b) ;
    \draw [->] (b) -- node[above] {2} (c) ;
    \draw [->] (a) -- node[above] {2} (c) ;
\end{tikzpicture}
\end{center}
\end{figure}

\textbf{Figure 3.1}
that \( f_2 \) holds at the last state of the prefix and \( f_1 \) holds at all other states along the prefix.

Formally, we define the semantics of CTL formulae with respect to a structure \( M = (S, \mathcal{A}_1, \ldots, \mathcal{A}_k, \mathcal{I}) \) which consists of

- \( S \) - a countable set of states,
- \( \mathcal{A}_i \subseteq S \times S \), a binary relation on \( S \) giving the possible transitions by process \( i \), and
- \( \mathcal{I} \) - an assignment of atomic propositions true in each state.

Let \( A = A_1 \cup \ldots \cup A_k \). We require that \( A \) be total, i.e., that \( \forall x \in S \exists y (x, y) \in A \). A path is an infinite sequence of states \( (s_0, s_1, s_2, \ldots) \in S \) such that \( \forall i (s_i, s_{i+1}) \in A \). To any structure \( M \) and state \( s \in S \) of \( M \), there corresponds a computation tree with root labelled \( s_0 \) such that \( s_0 \models t \) is an arc in the tree iff \( (s, t) \in A_1 \). See Figure 3.1.

We use the usual notation to indicate truth in a structure: \( M, s_0 \models f \) means that at state \( s_0 \) in structure \( M \) formula \( f \) holds true. When the structure \( M \) is understood, we write \( s_0 \models f \). We define \( \models \) inductively:

\[
\begin{align*}
  s_0 \models p & \quad \text{iff } p \in \mathcal{I}(s_0) \\
  s_0 \models \neg f & \quad \text{iff not } (s_0 \models f) \\
  s_0 \models f_1 \land f_2 & \quad \text{iff } s_0 \models f_1 \text{ and } s_0 \models f_2 \\
  s_0 \models \exists j f & \quad \text{iff for some state } t \text{ such that } (s_0, t) \in A_j, t \models f \\
  s_0 \models A[f_1 U f_2] & \quad \text{iff for all paths } (s_0, s_1' \ldots), \exists i [i \geq 0 \land s_i \models f_2 \\
  \land \forall j (0 \leq j \land j < i \rightarrow s_j \models f_1)] \\
  s_0 \models E[f_1 U f_2] & \quad \text{iff for some path } (s_0, s_1' \ldots), \exists i [i \geq 0 \land s_i \models f_2 \\
  \land \forall j (0 \leq j \land j < i \rightarrow s_j \models f_1)]
\end{align*}
\]
We write $\vdash f$ to indicate that $f$ is universally valid, i.e., true at all states in all structures. Similarly, we write $\models f$ to indicate that $f$ is satisfiable, i.e., $f$ is true in some state of some structure.

We introduce some abbreviations:

$f_1 \vee f_2 \equiv (\neg f_1 \wedge \neg f_2)$, $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$, and $f_1 \leftrightarrow f_2 \equiv (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$

for logical disjunction, implication, and equivalence, respectively.

$A[f_1, f_2] \equiv \neg E[\neg f_1 \cup \neg f_2]$ which means for every path, for every state $s$ on the path, if $f_1$ is false at all states on the path prior to $s$, then $f_2$ holds at $s$.

$E[f_1, f_2] \equiv \neg A[\neg f_1 \cup \neg f_2]$ which means for some path, for every state $s$ on the path, if $f_1$ is false at all states on the path prior to $s$, then $f_2$ holds at $s$.

$\text{AFF}_{f_1} \equiv A[\text{true} \cup f_1]$ which means for every path, there exists a state on the path at which $f_1$ holds.

$\text{EFF}_{f_1} \equiv E[\text{true} \cup f_1]$ which means for some path, there exists a state on the path at which $f_1$ holds.

$\text{AG}_{f_1} \equiv \neg E \neg f_1$ which means for every path, at every node on the path $f_1$ holds.

$\text{EG}_{f_1} \equiv \neg A \neg f_1$ which means for some path, at every node on the path $f_1$ holds.

$\text{AX}_{i}f \equiv \neg \text{EX}_{i} \neg f$ which means at all successor states reachable by an atomic step of process $P_i$, $f$ holds.

$\text{EX}_{i}f \equiv \text{EX}_{1}f \lor \ldots \lor \text{EX}_{k}f$ which means at some successor state $f$ holds.

$\text{AXf} \equiv \neg \text{EXf}$ which means at all successor states $f$ holds.

See Fig. 3.2 for illustrations of some of the above modalities.
4. FIXPOINT CHARACTERIZATIONS

Each of the modal operators such as AU, EG, EF, etc., may be characterized as an extremal fixpoint of an appropriate monotonic functional. Let $M = (S, A_1, \ldots, A_k, \phi)$ be an arbitrary structure. We use PRED(S) to denote the lattice of total predicates over $S$ where each predicate is identified with the set of states which make it true and the ordering is set inclusion. Then, each formula $f$ defines a member of $\text{PRED}(S) = \{s: M, s \models f\}$. Let $\tau : \text{PRED}(S) \rightarrow \text{PRED}(S)$ be given; then

1. $\tau$ is monotonic provided that $P \subseteq Q$ implies $\tau[P] \subseteq \tau[Q]$;

2. $\tau$ is $U$-continuous provided that $P_1 \subseteq P_2 \subseteq \ldots$ implies $\tau[\bigcup_i P_i] = \bigcup_i \tau[P_i]$;

3. $\tau$ is $\cap$-continuous provided that $P_1 \supseteq P_2 \supseteq \ldots$ implies $\tau[\bigcap_i P_i] = \bigcap_i \tau[P_i]$. [1]

A monotonic functional $\tau$ on $\text{PRED}(S)$ always has both a least fixpoint, $\text{lfp}X.\tau[X]$, and a greatest fixpoint, $\text{gfp}X.\tau[X]$ (see Tarski [TA55]):

$\text{lfp}X.\tau[X] = \bigcap \{X : \tau[X] = X\}$ whenever $\tau$ is monotonic, and $\text{lfp}X.\tau[X] = \bigcup_i \tau^i[\text{False}]$ whenever $\tau$ is also $U$-continuous; $\text{gfp}X.\tau[X] = \bigcup \{X : \tau[X] = X\}$ whenever $\tau$ is monotonic, and $\text{gfp}X.\tau[X] = \bigcap_i \tau^i[\text{True}]$ whenever $\tau$ is also $\cap$-continuous.

The modal operators have the following fixpoint characterization:

$$\text{EF}h = \text{lfp}Z. h \lor \text{EX}Z$$

$$\text{AF}h = \text{lfp}Z. h \lor \text{AX}Z$$

$$\text{E}[gU]h = \text{lfp}Z. h \lor (g \land \text{EX}Z)$$

$$\text{A}[gU]h = \text{lfp}Z. h \lor (g \land \text{AX}Z)$$
AGh = gfpZ.h ∧ AXZ
EGh = gfpZ.h ∧ EXZ
E[gVh] = gfpZ.h ∧ (g ∨ EXZ)
A[gVh] = gfpZ.h ∧ (g ∨ AXZ)

If all $A_i$ in $M$ are of bounded nondeterminism, then each of the functional used in the fixpoint characterizations above is $U$-continuous and $∅$-continuous as well as monotonic. We show that the first fixpoint characterization is correct:

PROPOSITION. 4.1. $EFh$ is the least fixpoint of the functional $τ[Z] = h ∨ EXZ$.

Proof. We first show that $EFh$ is a fixpoint of $τ[Z]$: Suppose $s_0 ⊨ EFh$. Then by definition of $|−|$, there is a path $(s_0, s_1, s_2, ...) \in M$ such that for some $k$, $s_k ⊨ EFh$. If $k = 0$, $s_0 ⊨ h$. Otherwise $s_1 ⊨ EFh$ and $s_0 ⊨ EXEFh$. Thus, $EFh ⊨ h ∨ EXEFh$. Similarly, if $s_0 ⊨ h ∨ EXEFh$, then $s_0 ⊨ h$ or $s_0 ⊨ EXEFh$. In either case, $s_0 ⊨ EFh$ and $h ∨ EXEFh ⊨ EFh$. Thus $EFh = h ∨ EXEFh$.

To see that $EFh$ is the least fixpoint of $τ[Z]$, it suffices to show that $EFh = \bigcup_{i \geq 0} τ^i[False]$. It follows by a straightforward induction on $i$ that $s_0 \in τ^i[False]$ iff there is a finite path $(s_0, s_1, ..., s_i)$ in $M$ and a $j \leq i$ for which $s_j ⊨ h$.

These fixpoint characterizations are helpful in proving the correctness of the model checking algorithm of Section 5 and are also used in constructing the tableau for the decision procedure of Section 6. Fixpoint characterizations have been investigated, in other contexts, by a number of researchers including [PA69], [CL77], [FS81], and [EC80].
5. MODEL CHECKER

Assume that we wish to determine whether formula \( f \) is true in the finite structure \( M = (S, A_1, \ldots, A_k, \mathcal{Q}) \). Let \( \text{sub}^+(f_0) \) denote the set sub-formulae of \( f_0 \) with main connective other than \( \sim \). We label each state \( s \in S \) with the set of positive/negative formulae \( f \) in \( \text{sub}^+(f_0) \) so that

\[
\begin{align*}
 f & \in \text{label}(s) \quad \text{iff} \quad M, s \models f \\
\sim f & \in \text{label}(s) \quad \text{iff} \quad M, s \models \sim f .
\end{align*}
\]

The algorithm makes \( n + 1 \) passes where \( n = \text{length}(f_0) \). On pass \( i \), every state \( s \in S \) is labelled with \( f \) or \( \sim f \) for each formula \( f \in \text{sub}^+(f_0) \) of length \( i \). Information gathered in earlier passes about formulae of length less than \( i \) is used to perform the labelling. For example, if \( f = f_1 \land f_2 \), then \( f \) should be placed in the set for \( s \) precisely when \( f_1 \) and \( f_2 \) are already present in the set for \( s \). For modalities such as \( \mathcal{A}[f_1 \mathcal{U} f_2] \) information from the successor states of \( s \) (as well as from \( s \) itself) is used. Since \( \mathcal{A}[f_1 \mathcal{U} f_2] = f_2 \lor (f_1 \land \mathcal{A} \mathcal{X} \mathcal{A}[f_1 \mathcal{U} f_2]) \), \( \mathcal{A}[f_1 \mathcal{U} f_2] \) should be placed in the set for \( s \) when \( f_2 \) is already in the set for \( s \) or when \( f_1 \) is in the set for \( s \) and \( \mathcal{A}[f_1 \mathcal{U} f_2] \) is in the set of each immediate successor state of \( s \).

Satisfaction of \( \mathcal{A}[f_1 \mathcal{U} f_2] \) may be seen to "radiate" outward from states where it holds immediately by virtue of \( f_2 \) holding:

Let \[
\begin{align*}
(A[f_1 \mathcal{U} f_2])^0 &= f_2 \\
(A[f_1 \mathcal{U} f_2])^{k+1} &= f_2 \lor \mathcal{A} \mathcal{X} (A[f_1 \mathcal{U} f_2])^k .
\end{align*}
\]

It can be shown that \( M, s \models (A[f_1 \mathcal{U} f_2])^k \) iff \( M, s \models A[f_1 \mathcal{U} f_2] \) and along every path starting at \( s \), \( f_2 \) holds by the \( k \)th state following \( s \). Thus,
states where \((A[f_1Uf_2])^0\) holds are found first, then states where
\((A[f_1Uf_2])^1\) holds, etc. If \(A[f_1Uf_2]\) holds, then \((A[f_1Uf_2])^{\text{card}(S)}\)
must hold since all loop-free paths in \(M\) are of length \(\leq\text{card}(S)\). Thus,
if after \(\text{card}(S)\) steps of radiating outward, \(A[f_1Uf_2]\) has still not been
found to hold at state \(s\), then put \(\neg A[f_1Uf_2]\) in the set for \(s\).

The algorithm for pass \(i\) is listed below in an Algol-like syntax:

\[
\text{for every state } s \in S \text{ do}
\quad \text{for every } f \in \text{sub}^+(f_0) \text{ of length } i \text{ do}
\quad \quad \text{if } f = A[f_1Uf_2] \text{ and } f_2 \in \text{set}(s) \text{ or}
\quad \quad \quad f = E[f_1Uf_2] \text{ and } f_2 \in \text{set}(s) \text{ or}
\quad \quad \quad f = \exists \text{EX}_j f_1 \text{ and } \exists t((s,t) \in A_j \text{ and } f_1 \in \text{set}(t)) \text{ or}
\quad \quad \quad f = f_1 \land f_2 \text{ and } f_1 \in \text{set}(s) \text{ and } f_2 \in \text{set}(s)
\quad \quad \text{then add } f \text{ to set}(s)
\quad \text{end}
\text{end;}
\]

\[
A: \text{for } j = 1 \text{ to card}(s) \text{ do}
\quad \text{for every state } s \in S \text{ do}
\quad \quad \text{for every } f \in \text{sub}^+(f_0) \text{ of length } i \text{ do}
\quad \quad \quad \text{if } f = A[f_1Uf_2] \text{ and } f_1 \in \text{set}(s) \text{ and}
\quad \quad \quad \quad \forall t((s,t) \in A \land f \in \text{set}(t)) \text{ or}
\quad \quad \quad \quad f = E[f_1Uf_2] \text{ and } f_1 \in \text{set}(s) \text{ and}
\quad \quad \quad \quad \exists t((s,t) \in A \land f \in \text{set}(t))
\quad \quad \text{then add } f \text{ to set}(s)
\quad \text{end}
\text{end;}
\]

\[
B: \text{end}
\text{end;}
\]

\[
\text{for every state } s \in S \text{ do}
\quad \text{for every } f \in \text{sub}^+(f_0) \text{ of length } i \text{ do}
\quad \quad \text{if } f \notin \text{set}(s)
\quad \quad \text{then add } \neg f \text{ to set}(s)
\quad \text{end}
\text{end}
\]

C: end
Figures 5.1-5.5 give snapshots of the algorithm in operation on the structure shown for the formula $AFb \land EGa$ (which abbreviates $AFb \land \neg AF\neg a$).

Suppose we extend the logic to permit $\forall$ path $\forall$ node $p$ or, equivalently, its dual $\exists$ path $\exists$ node $P$ which we write $\mathcal{E}P$. We can generalize the model checker to handle this case by using the following proposition:

**Proposition 5.1.** Let $M = (S, A_1, \ldots, A_k, \mathcal{L})$ be a structure and $s \in S$. Then $M, s \models \mathcal{E}P$ iff there exists a path from $s$ to a node $s'$ such that $M, s' \models p$ and either $s'$ is a successor of itself or the strongly connected component of $M$ containing $s'$ has cardinality greater than 1 (see Fig. 5.6).

**Proof.** (only if:) Suppose $M, s \models \mathcal{E}P$. Then there is an infinite path $(s_0, s_1, s_2, \ldots)$ through $M$ and a state $s' \in S$ such that

1. $s_0 = s$;
2. $s' = s_i$ for infinitely many distinct $i$;
3. $M, s' \models p$.

If $s'$ is a successor of itself, we are done. Otherwise, there is a finite path $(s', \ldots, s'', \ldots s')$ from $s'$ back to itself (because of (2)) which contains a state $s'' \neq s$. So, $s''$ is reachable from $s'$ and $s'$ is reachable from $s''$, and $s'$ is in a strongly connected component of $M$ of cardinality greater than 1.

(if:) If $s'$ is a successor of itself, then $p$ is true infinitely often along the path $(s', s', \ldots)$. Since $s'$ is reachable from $s$, $M, s \models \mathcal{E}P$. If the strongly connected component of $M$ containing $s'$ is of cardinality greater than 1, then there is a state $s'' \neq s'$ such that $s'$ is reachable from $s''$ and $s''$ is reachable from $s'$. Hence there is a finite path from $s'$ back to itself, and an infinite path starting
1st time at label A in pass 1

Figure 5.1
1st time at label B in pass 1

Figure 5.2
2nd time at label B in pass 1

Figure 5.3
1st time at label C in pass 1

Figure 5.4
at termination

Figure 5.5
Testing for $\mathsf{EF_p}$

Figure 5.6
at \( s' \) which goes through \( s' \) infinitely often. Since \( s' \) is reachable from \( s \), \( M, s \models \mathsf{E FP} \).

Notice that all algorithms discussed so far run in time polynomial in the size of the candidate model and formula. The algorithm for basic CTL presented above runs in time \( \text{length}(f) \cdot (\text{card}(s))^2 \). Since there is a linear time algorithm for finding the strongly connected components of a graph [TA72], we can also achieve the \( \text{length}(f) \cdot (\text{card}(S))^2 \) time bound when we include the infinitary quantifiers.

Finally, we show that it is not always possible to obtain polynomial time algorithms for model checking. Suppose we extend our language to allow either an existential or a universal path quantifier to prefix an arbitrary assertion from linear time logic as in [LA80] and [GP80]. Thus, we can write assertions such as

\[
E[F_{q_1} \wedge \ldots \wedge F_{q_n} \wedge G_{h_1} \wedge \ldots \wedge G_{h_n}]
\]

meaning

"there exists a computation path \( \rho \) such that, along \( \rho \)

sometimes \( q_1 \) and \( \ldots \) and sometimes \( q_n \) and

always \( h_1 \) and \( \ldots \) and always \( h_n \)."

We claim that the problem of determining whether a given formula \( f \) holds in a given finite structure \( M \) is \( \mathsf{NP} \)-hard.

**Theorem 5.2.** Directed Hamiltonian Path is reducible to the problem of determining whether \( M, s \models f \) where

- \( M \) is a finite structure,
- \( s \) is a state in \( M \) and
- \( f \) is the assertion (using atomic propositions \( p_1, \ldots, p_n \)):

\[
E[F_{p_1} \wedge \ldots \wedge F_{p_n} \wedge G(p_1 \rightarrow XG p_1) \wedge \ldots \wedge G(p_n \rightarrow XG \sim p_n)]
\]
Proof. Consider an arbitrary directed graph $G = (V, A)$ where $V = \{v_1, \ldots, v_n\}$. We obtain a structure from $G$ by making proposition $p_i$ hold at node $v_i$ and false at all other nodes (for $1 \leq i \leq n$), and by adding a source node $u_1$ from which all $v_i$ are accessible (but not vice versa) and a sink node $u_2$ which is accessible from all $v_i$ (but not vice versa).

Formally, let the structure $M = (U, B, \mathcal{L})$ consist of

$$U = V \cup \{u_1, u_2\} \text{ where } u_1, u_2 \notin V$$

$\mathcal{L}$, on assignment of states to propositions such that

$$v_i \models p_i, v_i \not\models p_j, (1 \leq i, j \leq n, i \neq j)$$

$$u_1 \not\models p_i, u_2 \not\models p_i, (1 \leq i \leq n) \text{ and}$$

$$B = A \cup \{(u_1, v_i) : v_i \in V\} \cup \{(v_i, u_2) : v_i \in V\} \cup \{(u_2, u_2)\}.$$

It follows that

$$M, u_1 \models f \text{ iff there is a directed infinite path in } M$$

starting at $u_1$ which goes through all $v_i \in V$

exactly once and ends in the self-loop through $u_2$;

iff there is a directed Hamiltonian path

in $G$. □

We believe that the model checker may turn out to be of considerable value in the verification of certain finite state concurrent systems such as network protocols. We have developed an experimental implementation of the model checker at Harvard which is written in C and runs on the DEC 11-70.
6. THE DECISION PROCEDURE

In this section we outline a tableau-based decision procedure for satis-
fiability of CTL formulae. Our algorithm is similar to one proposed for
UB in [BM81]. Tableau-based decision procedures for simpler program logics
such as PDL and DPDL are given in [PR77] and [BH81]. The reader should
consult [HC68] for a discussion of tableau-based decision procedures for
classical modal logics and [SM68] for a discussion of tableau-based decision
procedures for propositional logic.

We now briefly describe the decision procedure for CTL and
illustrate it with a simple example. The decision procedure is described
in detail in the appendix. To simplify the notation in the present discus-
sion, we omit the labels on arcs which are normally used to distinguish
between transitions by different processes.

The decision procedure takes as input a formula $f_0$ and returns either
"YES, $f_0$ is satisfiable," or "NO, $f_0$ is unsatisfiable." If $f_0$ is satis-
fiably, a finite model is constructed. The decision procedure performs the
following steps:

1. Build the initial tableau $T$ which encodes potential models of
   $f_0$. If $f_0$ is satisfiable, it has a finite model that can be
   "embedded" in $T$.

2. Test the tableau for consistency by deleting inconsistent portions.
   If the "root" of the tableau is deleted, $f_0$ is unsatisfiable.
   Otherwise, $f_0$ is satisfiable.

*The [BM81] algorithm is incorrect and will erroneously claim that certain
satisfiable formulae are unsatisfiable. Correct tableau-based and filtration-
based decision procedures for UB are given in [EH81]. In addition, Ben-Ari
[BA81] states that a corrected version, using different techniques, of [BM81]
is forthcoming.
3. Unravel the tableau into a model of $f_0$.

The decision procedure begins by building a tableau $T$ which is a finite directed AND/OR graph. Each node of $T$ is either an AND-node or an OR-node and is labelled by a set of formulae. We use $G_1,G_2,...$ to denote the labels of OR-nodes, $H_1,H_2,...$ to denote the labels of AND-nodes, and $F_1,F_2,...$ to denote the labels of arbitrary nodes of either type. No two AND-nodes have the same label, and no two OR-nodes have the same label. The intended meaning is that, when node $F$ is considered as a state in an appropriate structure, $F \models f$ for all $f \in F$. The tableau $T$ has a "root" node $G_0 = \{f_0\}$ from which all other nodes in $T$ are accessible.

The set of successors of an OR-node $G$, Blocks($G$) = $\{H_1,H_2,...,H_k\}$ has the property that

$$\models G \iff \models H_1 \text{ or } ... \text{ or } \models H_k.$$ 

We can explain the construction of Blocks($G$) as follows: Each formula in $G$ may be viewed as a conjunctive formula $\alpha \equiv \alpha_1 \land \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \lor \beta_2$. Clearly, $f \land g$ is an $\alpha$ formula and $f \lor g$ is a $\beta$ formula. A modal formula may be classified as $\alpha$ or $\beta$ based on its fixpoint characterization; thus, $EFP = p \lor E\!X\!EFP$ is a $\beta$ formula and $AGp = p \land A\!X\!AGp$ is an $\alpha$ formula. A formula that involves no modalities or has main connective one of EX or AX is both $\alpha$ and $\beta$ and is called an elementary formula. Any other formula is nonelementary. We say that a set of formulae $F$ is downward closed provided that (i) if $\alpha \in F$ then $\alpha_1, \alpha_2 \in F$, and (ii) if $\beta \in F$ then $\beta_1 \in F$ or $\beta_2 \in F$. We construct the members $H_i$ of Blocks($G$) by repeatedly expanding each nonelementary formula in $G$ into its $\alpha$ or $\beta$ components. Each $\beta$ expansion results in two blocks, one which will contain $\beta_1$ and the other which will contain $\beta_2$. Expansion stops when all $H_i$ are downward closed.
Blocks \( \mathcal{S}_0 = \{ \mathcal{H}_0, \ldots, \mathcal{H}_3 \} \). Each \( \mathcal{H}_i \) is a downward closed set containing \( \mathcal{S}_0 \) and is obtained by taking the union of all formulae occurring along the path from the root to the \( i \)-th leaf of the \( \alpha\beta \)-expansion tree of \( EFp \land EF\sim p \).

Figure 6.1
The set of successors of an AND-node \( H \), \( \text{Tiles}(H) = \{ G_1, G_2, \ldots, G_k \} \) has the property that, if \( H \) contains no propositional inconsistencies, then

\[ \models H \iff \models G_1 \text{ and } \ldots \text{ and } \models G_k. \]

To construct \( \text{Tiles}(H) \) we use the information supplied by the elementary formulae in \( H \). For example, if \( \{ \text{AX}h_1, \text{AX}h_2, \text{EX}g_1, \text{EX}g_2, \text{EX}g_3 \} \) is the set of all elementary formulae in \( H \), then \( \text{Tiles}(G) = \{ \{ h_1, h_2, g_1 \}, \{ h_1, h_2, g_2 \}, \{ h_1, h_2, g_3 \} \} \).

To build \( T \), we start out by letting \( G_0 = \{ f_0 \} \) be the root node. Then we create \( \text{Blocks}(G_0) = \{ H_1, H_2, \ldots, H_k \} \) and attach each \( H_i \) as a successor of \( G_0 \). For each \( H_i \) we create \( \text{Tiles}(H_i) \) and attach its members as the successors of \( H_i \). For each \( G_j \in \text{Tiles}(H_i) \) we create \( \text{Blocks}(G_j) \), etc. Whenever we encounter two nodes of the same type with identical labels we identify them. This ensures that no two AND-nodes will have the same label, and that no two OR-nodes will have the same label. The tableau construction will eventually terminate since there are only \( 2^{\text{length}(f_0)} \) possible labels each of which can occur at most twice.

Suppose, for example, that we want to determine whether \( \text{EF}_{\text{F}} \land \text{EF}_{\neg \text{F}} \) is satisfiable. We build the tableau \( T \), starting with root node \( G_0 = \{ \text{EF}_{\text{F}}, \text{EF}_{\neg \text{F}} \} \). We construct \( \text{Blocks}(G_0) = \{ H_0, H_1, H_2, H_3 \} \) as shown in Figure 6.1. Each \( H_i \) is attached as a successor of \( G_0 \). Next, \( \text{Tiles}(H_i) \) is determined for each \( H_i \) (except \( H_1 \) which is immediately seen to contain a propositional inconsistency) and its members are attached as successors of \( H_i \). (Note that two copies of \( G_1 = \{ \text{EF}_{\neg \text{F}} \} \) are created, one in \( \text{Tiles}(H_0) \) and the other in \( \text{Tiles}(H_2) \); but they are then merged into a single node.) Similarly, \( G_2 \in \text{Tiles}(H_2) \cap \text{Tiles}(H_3) \). Continuing in this fashion we obtain the complete tableau shown in Fig. 6.2.
Figure 6.2
Next we must test the tableau for consistency. Note that $H_1$ is inconsistent because it contains both $p$ and $\neg p$. We must also check that it is possible for eventuality formulae such as $AFh$ or $EFh$ to be fulfilled: e.g., if $EFh \in F$, then there must be some node $F'$ reachable from $F$ such that $h \in F'$. If any node fails to pass this test, it is marked inconsistent. In this example, all nodes pass the test. Since the root is not marked inconsistent, $EFp \land EP\neg p$ is satisfiable.

Finally, we construct a model $M$ of $EFp \land EP\neg p$. The states in $M$ will be (copies of) the AND-nodes in the tableau. The model will have the property that for each state $H, M, H \models f$ for all $f \in M$. The root of $M$ can be any consistent state $H_1 \in Blocks(G_0)$. We choose $H_0$. Now $H_0$ contains the eventualities $EFp$ and $EF\neg p$. We must ensure that they are actually fulfilled in $M$. $EFp$ is immediately fulfilled in $H_0$, but $EF\neg p$ is not. So when we choose a successor state to $H_0$, which must be one of $H_4$ or $H_5$, we want to ensure that $EF\neg p$ is fulfilled. Thus, we choose $H_5$. Finally, the only possible successor state of $H_5$ is $H_5$ itself. We obtain the model shown in Fig. 6.3 which is embedded in the tableau.
Figure 6.3
7. SYNTHESIS ALGORITHM

We now present our method of synthesizing synchronization skeletons from a CTL description of their intended behavior. We identify the following steps:

1. Specify the desired behavior of the concurrent system using CTL.
2. Apply the decision procedure to the resulting CTL formula in order to obtain a finite model of the formula.
3. Factor out the synchronization skeletons of the individual processes from the global system flowgraph defined by the model.

We illustrate the method by solving a mutual exclusion problem for processes $P_1$ and $P_2$. Each process is always in one of three regions of code:

- **NCS$_i$** the NonCritical Section
- **TRY$_i$** the TRYing Section
- **CS$_i$** the Critical Section

which it moves through as suggested in Fig. 7.1.

When it is in region NCS$_i$, process $P_i$ performs "noncritical" computations which can proceed in parallel with computations by the other process $P_j$. At certain times, however, $P_i$ may need to perform certain "critical" computations in the region CS$_i$. Thus, $P_i$ remains in NCS$_i$ as long as it has not yet decided to attempt critical section entry. When and if it decides to make this attempt, it moves into the region TRY$_i$. From there it enters CS$_i$ as soon as possible, provided that the mutual exclusion constraint $\sim (C_{S_1} \land C_{S_2})$ is not violated. It remains in CS$_i$ as long as necessary to perform its "critical" computations and then re-enters NCS$_i$. 
Figure 7.1
Note that in the synchronization skeleton described, we only record transitions between different regions of code. Moves entirely within the same region are not considered in specifying synchronization. Listed below are the CTL formulae whose conjunction specifies the mutual exclusion system:

1. start state
   \[ NCS_1 \land NCS_2 \]

2. mutual exclusion
   \[ \text{AG}(\neg(CS_1 \land CS_2)) \]

3. absence of starvation for \( P_i \)
   \[ \text{AG}(\text{TRY}_i \rightarrow \text{AFCS}_i) \]

4. each process \( P_i \) is always in exactly one of the three code regions
   \[ \text{AG}(NCS_i \lor \text{TRY}_i \lor \text{CS}_i) \]
   \[ \text{AG}(NCS_i \rightarrow (\text{TRY}_i \lor \text{CS}_i)) \]
   \[ \text{AG}(\text{TRY}_i \rightarrow (NCS_i \lor \text{CS}_i)) \]
   \[ \text{AG}(\text{CS}_i \rightarrow (NCS_i \lor \text{TRY}_i)) \]

5. it is always possible for \( P_i \) to enter its trying region from its noncritical region
   \[ \text{AG}(NCS_i \rightarrow \text{EX}_i \text{TRY}_i) \]

6. it is always the case that any move \( P_i \) makes from its trying region is into the critical region
   \[ \text{AG}(\text{TRY}_i \land \text{EX}_i \text{True} \rightarrow \text{AX}_i \text{CS}_i) \]

7. it is always possible for \( P_i \) to re-enter its noncritical region from its critical region
   \[ \text{AG}(	ext{CS}_i \rightarrow \text{EX}_i \text{NCS}_i) \]

8. a transition by one process cannot cause a move by the other
   \[ \text{AG}(NCS_i \rightarrow \text{AX}_j \text{NCS}_j) \]
   \[ \text{AG}(\text{TRY}_i \rightarrow \text{AX}_j \text{TRY}_j) \]
   \[ \text{AG}(\text{CS}_i \rightarrow \text{AX}_j \text{CS}_j) \]

9. some process can always move
   \[ \text{AG}(\text{EX}_i \text{True}) \]

We must now construct the initial AND/OR graph tableau. In order to reduce the recording of inessential or redundant information in the node
labels we observe the following rules:

(1) Automatically convert a formula of the form \( f_1 \wedge \ldots \wedge f_n \) to the set of formulae \( \{f_1, \ldots, f_n\} \). (Recall that the set of formulae \( \{f_1, \ldots, f_n\} \) is satisfiable iff \( f_1 \wedge \ldots \wedge f_n \) is satisfiable.)

(2) Do not physically write down an invariance assertion of the form \( \text{AGf} \) because it holds everywhere as do its consequences \( f \) and \( \text{AXAGf} \) (obtained by \( \alpha \)-expansion). The consequence \( \text{AXAGf} \) serves only to propagate forward the truth of \( \text{AGf} \) to any "descendent" nodes in the tableau. Do that propagation automatically but without writing down \( \text{AGf} \) in any of the descendent nodes. The consequence \( f \) may be written down if needed.

(3) An assertion of the form \( f \lor g \) need not be recorded when \( f \) is already present. Since any state which satisfies \( f \) must also satisfy \( f \lor g \), \( f \lor g \) is redundant.

(4) If we have \( \text{TRY}_i \) present, there is no need to record \( \sim \text{NCS}_i \) and \( \sim \text{CS}_i \). If we have \( \text{NCS}_i \) present, there is no need to record \( \sim \text{TRY}_i \) and \( \sim \text{CS}_i \). If we have \( \text{CS}_i \) present, there is no need to record \( \sim \text{NCS}_i \) and \( \sim \text{TRY}_i \).

By the above conventions, the root node of the tableau will have the two formulae \( \text{NCS}_1 \) and \( \text{NCS}_2 \) recorded in its label which we now write as \(<\text{NCS}_1 \text{NCS}_2>\). In building the tableau, it will be helpful to have constructed \( \text{Blocks}(G) \) for the following OR-nodes: \(<\text{NCS}_1 \text{NCS}_2>\), \(<\text{TRY}_1 \text{NCS}_2>\), \(<\text{CS}_1 \text{NCS}_2>\), \(<\text{TRY}_1 \text{TRY}_2>\), and \(<\text{CS}_1 \text{TRY}_2>\). For all other OR-nodes \( G' \) appearing in the tableau, \( \text{Blocks}(G') \) will be identical to or can be obtained by symmetry from \( \text{Blocks}(G) \) for some \( G \) in the above list. Figures 7.2-7.6 show the abbreviated construction of \( \text{Blocks}(G) \) for these OR-nodes as well as \( \text{Tiles}(H) \) for each \( H \in \text{Blocks}(G) \). We then build the tableau using the
Figure 7.3
Figure 7.4
Figure 7.6
information about Blocks and Tiles contained in Figures 7.2-7.6. We next apply the marking rules to delete inconsistent nodes. Note that the OR-node $\langle CS_1 CS_2 AFCS_2 \rangle$ is marked as deleted because of a propositional inconsistency (with $\sim (CS_1 \land CS_2)$), a consequence of the unwritten invariance $AG(\sim (CS_1 \land CS))$. This, in turn, causes the AND-node that is the predecessor of $\langle CS_1 CS_2 AFCS_2 \rangle$ to be marked. The resulting tableau is shown in Fig. 7.7. Each node in Fig. 7.7 is labelled with a minimal set of formulae sufficient to distinguish it from any other node.

We construct a model $M$ from $T$ by pasting together model fragments for the AND-nodes using local structure information provided by $T$. Intuitively, a fragment is a rooted dag of AND-nodes embeddable in $T$ such that all eventuality formulae in the label of the root node are fulfilled in the fragment. Fragments are described in detail in the appendix.

The root node of the model is $H_0$, the unique successor of $G_0$. From the tableau we see that $H_0$ must have two successors, one of $H_1$ or $H_2$ and one of $H_3$ or $H_4$. Each candidate successor state contains an eventuality to fulfill, so we must construct and attach its fragment. Using the method described in the appendix, we choose the fragment rooted at $H_1$ to be the left successor and the fragment rooted at $H_4$ to be the right successor (see Fig. 7.8). This yields the portion of the model shown in Fig. 7.9.

We continue the construction by finding successors for each of the leaves: $H_5$, $H_9$, $H_{10}$ and $H_8$. We start with $H_5$. By inspection of $T$, we see that the only successors $H_5$ can have are $H_0$ and $H_9$. Since $H_0$ and $H_9$ already occur in the structure built so far, we add the arcs $H_5 \rightarrow H_0$ and $H_5 \rightarrow H_9$ to the structure. Note that this introduces a cycle $(H_0 \rightarrow H_1 \rightarrow H_5 \rightarrow H_0)$. In general, a cycle can be dangerous because it might
Figure 7.7
Figure 7.8

Figure 7.9
form a path along which some eventuality is never fulfilled; however, there
is no problem this time because the root of a fragment, $H_1$, occurs along the
cycle. A fragment root serves as a checkpoint to ensure that all eventuali-
ties are fulfilled. By symmetry between the roles of 1 and 2, we add in
the arcs $H_8 \xrightarrow{1} H_{10}$ and $H_8 \xrightarrow{2} H_0$. The structure now has the form shown in
Fig. 7.10.

We now have two leaves remaining: $H_9$ and $H_{10}$. We see from the
tableau that $H_4$ is a possible successor to $H_9$. We add in the arc
$H_9 \xrightarrow{1} H_4$. Again a cycle is formed but since $H_4$ is a fragment root no
problems arise. Similarly, we add in the arc $H_{10} \xrightarrow{2} H_1$. The decision pro-
cEDURE thus yields a model $M$ such that $M, s_0 \models f_0$ where $f_0$ is the con-
junction of the mutual exclusion system specifications. The model is shown
in Fig. 7.11 where only the propositions true in a state are retained in
the label.

We may view the model as a flowgraph of global system behavior. For
example, when the system is in state $H_1$, process $P_1$ is in its trying
region and process $P_2$ is in its noncritical section. $P_1$ may enter its
critical section or $P_2$ may enter its trying region. No other moves are
possible in state $H_1$. Note that all states except $H_6$ and $H_7$ are distin-
guished by their propositional labels. In order to distinguish $H_6$ from
$H_7$, we introduce a variable $\text{TUR}N$ which is set to 1 upon entry to $H_6$ and
to 2 upon entry to $H_7$. If we introduce $\text{TUR}N$'s value into the labels of
$H_6$ and $H_7$ then, the labels uniquely identify each node in the global
system flowgraph. See Fig. 7.12.

We describe how to obtain the synchronization skeletons of the indivi-
dual processes from the global system flowgraph. In the sequel we will
refer to these global system states by the propositional labels.
Figure 7.10
Figure 7.11
Figure 7.12
When $P_1$ is in $NCS_1$, there are three possible global states $[NCS_1 NCS_2]$ $[\text{TRY}_1 NCS_2]$ $[NCS_1 \text{CS}_2]$. In each case it is always possible for $P_1$ to make a transition into $\text{TRY}_1$ by the global transitions $[NCS_1 NCS_2] \xrightarrow{\text{TURN}=2} [\text{TRY}_1 NCS_2]$, $[NCS_1 \text{TRY}_2] \xrightarrow{\text{TURN}=2} [\text{TRY}_1 \text{TRY}_2]$, and $[NCS_1 \text{CS}_2] \xrightarrow{\text{TURN}=2} [\text{TRY}_1 \text{CS}_2]$. From each global transition by $P_1$, we obtain a transition in the synchronization skeleton of $P_1$. The $P_2$ component of the global state provides enabling conditions for the transitions in the skeleton of $P_1$. If along a global transition, there is an assignment to $\text{TURN}$, the assignment is copied into the corresponding transition of the synchronization skeleton. Thus we have the transitions shown in Fig. 7.13(a) in the synchronization skeleton of $P_1$. We merge the transitions which lack assignments to obtain the portion of the synchronization skeleton of $P_1$ shown in Fig. 7.13(b).

Now when $P_1$ is in $\text{TRY}_1$, there are four possible global states:

$[\text{TRY}_1 NCS_2]$, $[\text{TRY}_1 \text{TRY}_2 \text{TURN}=1]$, $[\text{TRY}_1 \text{TRY}_2 \text{TURN}=2]$, and $[\text{TRY}_1 \text{CS}_2]
$

and their associated global transitions by $P_1$:

$[\text{TRY}_1 NCS_2] \xrightarrow{1} [\text{CS}_1 NCS_2]$ and $[\text{TRY}_1 \text{TRY}_2 \text{TURN}=1] \xrightarrow{1} [\text{CS}_1 \text{TRY}_2]$. (No transitions by $P_1$ are possible in $[\text{TRY}_1 \text{TRY}_2 \text{TURN}=2]$ or $[\text{TRY}_1 \text{CS}_2]$.) Thus we obtain the portion of the synchronization skeleton for $P_1$ shown in Fig. 7.14(a). When $P_1$ is in $\text{CS}_1$ the associated global states and transitions are:

$[\text{CS}_1 NCS_2]$, $[\text{CS}_1 \text{TRY}_2]$, $[\text{CS}_1 NCS_2] \xrightarrow{1} [\text{NCS}_1 NCS_2]$, and $[\text{CS}_1 \text{TRY}_2] \xrightarrow{1} [\text{NCS}_1 \text{TRY}_2]$ from which we obtain the portion of the synchronization skeleton for $P_1$ shown in Fig. 7.14(b). Altogether, the synchronization skeleton for $P_1$ is shown in Fig. 7.15(a). By symmetry in the global state diagram we obtain the synchronization skeleton for $P_2$ as shown in Fig. 7.15(b).

The general method of factoring out the synchronization skeletons of the individual processes may be described as follows: Take the model of the
Figure 7.13 (a)

Figure 7.13 (b)
Figure 7.14 (a)

Figure 7.14 (b)
Figure 7.15(a)

Figure 7.15(b)
specification formula and retain only the propositional formulae in the labels of each node. There may now be distinct nodes with the same label. Auxiliary variables are introduced to ensure that each node gets a distinct label: if label \( L \) occurs at \( n > 1 \) distinct nodes \( v_1, \ldots, v_n \), then for each \( v_i \), set \( L := i \) on all arcs coming into \( v_i \) and add \( L = i \) as an additional component to the label of \( v_i \). The resulting newly labelled graph is the global system flowgraph.

We now construct the synchronization skeleton for process \( P_i \) which has \( m \) distinct code regions \( R_1, \ldots, R_m \). Initially, the synchronization skeleton for \( P_i \) is a graph with \( m \) distinct nodes \( R_1, \ldots, R_m \) and no arcs. Draw an arc from \( R_j \) to \( R_k \) if there is at least one arc of the form \( L_j \rightarrow L_k \) in the global system flowgraph where \( R_j \) is a component of the label \( L_j \) and \( R_k \) is a component of the label \( L_k \). The arc \( R_j \rightarrow R_k \) is a transition in the synchronization skeleton and is labelled with the enabling condition

\[
\bigvee \{(s_1 \land \ldots \land s_p) : [R_j s_1 \ldots s_p] \xrightarrow{i} [R_k s_1 \ldots s_p] \text{ is an arc in the global system flowgraph}\}.
\]

Add \( L := n \) to the label of \( R_j \rightarrow R_k \) if some arc \( [R_j s_1 \ldots s_p] \xrightarrow{i,L := n} [R_k s_1 \ldots s_p] \) also occurs in the flowgraph.
8. CONCLUSION

We have shown that it is possible to automatically synthesize the synchronization skeleton of a concurrent program from a Temporal Logic specification. We believe that this approach may in the long run turn out to be quite practical. Since synchronization skeletons are, in general, quite small, the potentially exponential behavior of our algorithm need not be an insurmountable obstacle. Much additional research may be needed, however, to make the approach feasible in practice.

We have also described a model checking algorithm which can be applied to mechanically verify that a finite state concurrent program meets a particular Temporal Logic specification. We believe that practical software tools based on this technique could be developed in the near future. Indeed, we have already programmed an experimental implementation of the model checker on the DEC 11/70 at Harvard.* Certain applications seem particularly suited to the model checker approach to verification: One example is the problem of verifying the correctness of existing network protocols many of which are coded as finite state machines. We encourage additional work in this area.

*We would like to acknowledge Marshall Brinn who did the actual programming for our implementation of the model checker.
9. BIBLIOGRAPHY


10. APPENDIX

In this appendix, we describe the decision procedure for CTL in detail. We assume that the reader is familiar with the overview of the decision procedure given in Section 6. The proofs of correctness of the decision procedure and of the finite model property for CTL are similar to the corresponding proofs for UB. See [EH81].

10.1 Construction of the Initial AND/OR graph

We construct the initial AND/OR graph $T$ in stages by the method below:

1. Initially, let the "root" node of $T$ be the OR-node $G_0 = \{f_0\}$.

2. If all nodes in $T$ have successors, halt. Otherwise, let $F$ be any node without successors in $T$. If $F$ is an OR-node $G$, construct $\text{Blocks}(G) = \{H_1, \ldots, H_k\}$ and attach each $H_i$ as an immediate successor of $G$ in $T$. If any $H_i$ has the same label as another AND-node $H$ already present in $T$, then merge $H_i$ and $H$. If $F$ is an AND-node $H$, construct $\text{Tiles}(H) = \{G_1, \ldots, G_k\}$ and attach each $G_i$ as an immediate successor of $G$ in $T$. Label the arc $(H, G_i)$ in $T$ with each $j$ such that $G_i \in \text{Tiles}_j(H)$. If any $G_i$ has the same label as some other OR-node $G$ already present in $T$, then merge $G_i$ and $G$. Repeat this step.

10.2 Construction of Blocks(G)

For convenience, we assume that every formula in $G$ has been placed in standard form with all negations driven inside so that only atomic propositions appear negated. (This can be done using duality: $\sim (f \land g) \leftrightarrow f \lor \sim g,$
We say that a formula is elementary provided that it is a proposition, the negation of a proposition, or has main connective \( \land \) or \( \lor \). Any other formula is nonelementary.

We classify nonelementary formulae as either \( \alpha \) or \( \beta \) as discussed in Section 6. The following table summarizes the classification:

\[
\begin{align*}
\alpha &= f \land g & \alpha_1 &= f & \alpha_2 &= g \\
\alpha &= A[f\lor g] & \alpha_1 &= g & \alpha_2 &= f \lor AXA[f\lor g] \\
\alpha &= E[f\lor g] & \alpha_1 &= g & \alpha_2 &= f \lor EXA[f\lor g] \\
\beta &= f \lor g & \beta_1 &= f & \beta_2 &= g \\
\beta &= A[f\land g] & \beta_1 &= g & \beta_2 &= f \land AXA[f\land g] \\
\beta &= E[f\land g] & \beta_1 &= g & \beta_2 &= f \land EXE[f\land g]
\end{align*}
\]

To construct Blocks\((G)\) we first build a finitely branching tree whose nodes are labelled with sets of formulae. (This tree is essentially a propositional logic tableau as described in [Smullyan].) Initially, let the root = \( G \). In general, let \( F \) be a leaf in the tree constructed so far for which there exists a nonelementary formula \( f \in F \). Add one or two sons to \( F \) as appropriate according to the rules shown in Fig. 10.1. Eventually, this construction must halt because all leaves \( F_1, \ldots, F_m \) will contain only elementary formulae. (This can be proved by induction of the length of the longest formula in \( G \).) Then let Blocks\((G) = \{H_1, \ldots, H_m\}\) where \( H_i \) is the set of all formulae appearing in some node on the path from \( F_i \) back to the root of the tree.

10.3 **Construction of Tiles\(_j\)(H)**

For each \( j \in [1:k] \), we must determine the set Tiles\(_j\)(H) of successors associated with process \( j \). Let
$\mathcal{F}$

$\mathcal{F}\setminus\{\alpha\} \cup \{\alpha_1, \alpha_2\}$

$\mathcal{F}\cup\{\beta\} \cup \{\beta_1\}$

$\mathcal{F}\cup\{\beta\} \cup \{\beta_2\}$

Figure 10.1
\[ \text{HA}_j = \{ f : \text{AX}_j \in \text{CH} \} \text{ and } \]
\[ \text{HE}_j = \{ g : \text{EX}_j g \in \text{CH} \}. \]

If \( \text{HE}_j \neq \emptyset \) then write \( \text{HE}_j \) as \( \{ g_1, \ldots, g_n \} \) and define
\[ \text{Tiles}_j(H) = \{ G_1^j, \ldots, G_n^j \} \text{ where } \]
\[ G_i^j = \text{HA}_j \cup \{ g_i \} \text{ for } i \in [1:n]. \]

Now define
\[ \text{Tiles}(H) = \bigcup \{ \text{Tiles}_j(H) : j \in [1:k] \}. \]

If \( G_i \in \text{Tiles}(H) \) then the arc from \( H \) to \( G_i \) in \( T \) is labelled with
\( j_1, \ldots, j_m \) where \( G_i \in \text{Tiles}_{j_1} (H), \ldots, \text{Tiles}_{j_m} (H) \). Figure 10.2 gives an example.

There are two special cases to consider. Let \( \text{HA} = \bigcup \{ \text{HA}_j : j \in [1:k] \} \)
and \( \text{HE} = \bigcup \{ \text{HE}_j : j \in [1:k] \} \). If \( \text{HA} \neq \emptyset \) and \( \text{HE} = \emptyset \) then split \( H \) into
\( H_1, \ldots, H_k \) where each \( H_j = H \cup \{ \text{EX}_j \text{True} \} \) and proceed as before. If
\( \text{HA} = \text{HE} = \emptyset \) the let \( \text{Tiles}(H) = \{ G \} \) where \( G = \{ f : f \in \text{CH} \} \) and let
\( \text{Blocks}(G) = \{ H \}. \)

10.4 Deleting Inconsistent Portions of the Tableau

We now apply the rules below to mark as inconsistent certain nodes of
the tableau \( T \). First we need the following definition:

A full subdag \( D \) rooted at node \( F \) in \( T \) is a finite, directed acyclic
subgraph of \( T \) satisfying the following 3 conditions:

1. For every OR-node \( G \in D \), there exists precisely one AND-node \( H \)
such that \( H \) is a son of \( G \) in \( D \) and in \( T \).
Figure 10.2
2. For every AND-node $H \in D$, if $H$ has any sons at all in $D$, then every son of $H$ in $T$ is a son of $H$ in $D$.

3. $F$ is the unique node in $D$ from which all other nodes are reachable.

Note that a full subdag $D$ is somewhat like a finite tree. It has a root (either an OR-node or an AND-node) and a frontier consisting of nodes with no successors in $D$ (although they may very well have successor when considered as nodes in $T$). All nodes of the frontier are AND-nodes.

Here are the marking rules:

markP: Mark as deleted any node $F$ which is immediately inconsistent, i.e., contains a formulae $f$ and its negation $\sim f$.

markOR: Mark as deleted any OR-node $G$ all of whose AND-node sons $H_i$ are already marked deleted.

markAND: Mark as deleted any AND-node $H$ one of whose OR-node sons $G_j$ is already marked deleted.

markEU: Mark as deleted any node $F$ such that $E[f_1 \cup f_2] \subseteq F$ and there does not exist some node $F'$ reachable from $F$ such that $f_2 \subseteq F'$ and for all $F''$ on some path from $F'$ back to $F, f_1 \subseteq F''$.

markAU: Mark as deleted any node $F$ such that $A[f_1 \cup f_2] \subseteq F$ and there does not exist a full subdag $D$ rooted at $F$ such that for all nodes $F'$ on the frontier of $D, f_2 \subseteq F'$ and for all non-frontier nodes $F''$ in $D, f_1 \subseteq F''$.

Apply the marking rules as long as possible. Marking must eventually stop because each successful application of a marking rule marks as deleted one node and there are only a finite number of nodes in $T$. 
If the root of $T$ is marked, then $f_0$ is unsatisfiable. If the root of $T$ is unmarked, then the subgraph of $T$ induced by the remaining unmarked nodes can be unraveled into a finite model of $f_0$.

10.5 Unravelling the Tableau into a Model

Let $T^*$ be the subgraph of $T$ that remains after all marked nodes and incident arcs have been deleted. We will construct a finite model $M$ of $f_0$ by "unravelling" $T^*$: For each AND-node $H$ in $T^*$, and for each eventuality formula $g \in H$, there is a full subdag rooted at $H$ which certifies that $g$ is fulfilled. (We know this subdag exists because $H$ is not marked by rule markAU or markEU on account of $g$.) We use these subdags to construct, for each AND-node $H$, a model fragment $MH$ such that every eventuality in $H$ is fulfilled within $MH$. We then splice together these fragments to obtain $M$.

10.6 Selecting Subdags

If $H$ is in $T^*$ and $g \in H$ is an eventuality formula, then there is a full subdag rooted at $H$ whose frontier nodes immediately fulfill $g$. There may be more than one such subdag. We wish to choose one of minimal size where the size of a subdag is the length of the longest path it contains. Our approach is to tag each node in $T^*$ with the size of the smallest subdag for $g$ rooted at the node.

Suppose, for example, that $g = A[fU_h]$. Initially, we set $\text{tag}(F) = 0$ for all nodes $F$ such that $h \in F$ and we set $\text{tag}(F) = \infty$ for all other nodes $F$. Then we let the size of full subdags radiate outward by making
card(T*) passes over the tableau. During each pass we perform the following step for each node F:

if F is an AND-node H such that A[fUH] ∈ H and tag(H) = ∞ and tag(G) < ∞ for all G ∈ Tiles(H) and f ∈ H
then let tag(H) := 1 + max{tag(G) : G ∈ Tiles(H)};
if F is an OR-node G such that A[fUH] ∈ G and tag(G) = ∞ and tag(H) < ∞ for some H ∈ Blocks(G)
then let tag(G) := min{tag(H) : H ∈ Blocks(G)};

After executing all card(T*) passes, if tag(F) = k < ∞ then there is a full subdag for g rooted at F of minimal size = k. To select a specific full subdag D we perform a construction in stages.

Initially let D₀ consist of the single node F.

In general, obtain Dᵢ₊₁ from Dᵢ as follows:

for all nodes F ∈ frontier(Dᵢ₊₁) do
if F is some OR-node G
then choose an AND-node H ∈ Blocks(G) with a minimal tag value
  (if there is more than one H eligible, choose one with a maximal card(Tiles(H)) value;
  if there is still more than one H eligible, choose the one of lowest index.)
attach H as the successor of G;
if F is some AND-node H
then add each member of Tiles(H) as a successor of F

Halt with D = Dᵢ when all frontier nodes of Dᵢ are AND-nodes H with tag(H) = 0. Let DAG[H,g] denote the subdag naturally induced by the AND-nodes of D. (Note: In the case where g = E[fUH], the construction of DAG[H,g] is similar.)
10.7 Construction of Fragments from Dags

For each AND-node H in $T^*$, we construct the fragment MH to have these properties:

1. MH is a dag consisting of (copies of) AND-nodes with root H.

2. MH is generated by $T^*$ in this sense: for all nodes $H_0$ in MH, if $\{H_1, \ldots, H_k\}$ is the set of successors of $H_0$ in MH, then there exist OR-nodes $G_1, \ldots, G_k$ in $T^*$ such that $Tiles(H_0) = \{G_1, \ldots, G_k\}$ and $H_i \in Blocks(G_i)$ for all $i \in [1:k]$. If the arc $(H_0, H_1)$ in MH has labels $j_1, \ldots, j_n$, then the arc $(H_0, G_i)$ has labels $j_1, \ldots, j_n$ in $T^*$.

3. All eventuality formulae in H are fulfilled in MH.

We construct MH in stages. Let $g_1, g_2, \ldots, g_m$ be a list of all eventuality formulae occurring in H. We build a sequence of dags $MH^1, \ldots, MH^m = MH$ so that, for each $j \in [1:m]$, $MH^j$ is a subgraph of $MH^{j+1}$ and $g_1, \ldots, g_j$ are fulfilled in $MH^j$.

Let $MH^1 = DAG[H, g_1]$. To obtain $MH^{i+1}$ from $MH^i$, do the following:

Identify any two nodes on $frontier(MH^i)$ with the same label; for all $H' \in frontier(MH^i)$ do

if $g_{i+1} \in H'$

then attach (a copy of) $DAG[H', g_{i+1}]$ to $MH^i$ at $H'$

end

Finally, let $MH = MH^m$.

10.8 Constructing the Model from Fragments

We construct M by splicing together fragments. Again, the construction is done in stages:
Let $M^1 = MH_0$ where $H_0 \in \text{Blocks}(\{f_0\})$ is chosen arbitrarily.

To construct $M^{i+1}$ from $M^i$ do the following:

for each $H \in \text{frontier}(M^i)$ do
  if there is a non-frontier node $H'$ that is the root of fragment $MH'$ in $M^i$ and has the same label as $H$
    then merge $H$ and $H'$
  else attach $MH$ to $M^i$ at $H$
end

The construction halts with $i = N$ when $\text{frontier}(M^N)$ is empty.

Let $M = M^N$.

THEOREM. The root of the fully marked tableau $T^*$ for CTL formula $f_0$ is unmarked iff $f_0$ is satisfiable. If $f_0$ is satisfiable, it is satisfiable in a finite model of size $O(\text{length}(f_0))$ for some $c > 1$.

The proof of this theorem will not be given; however, the proof of the corresponding theorem for UB is presented in [EH81].