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, USA

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

\[\text{This work was partially supported by NSF Grant MCS-7908365.}\]

© Springer-Verlag Berlin Heidelberg 2008

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 CSP-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. Sections 4 and 5 describe the model checker and the decision procedure, respectively. Finally, Section 6 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 $CS_i^j$ the $i$ 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 $CS_i^j$, 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 [BMB81]. 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 $\forall$ path $\exists$ node $P$ to express the inevitability of predicate $P$. The corresponding formula in our UB-like notation is $\text{AFP}$. The language presented in [EC80] is more expressive than UB as evidenced by the formula $\forall$ path $\forall$ 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 $f_i$ 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. $E_{i \leq j} f_1$ 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. $A[f_1 \land f_2]$ 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. $E[f_1 \land f_2]$ is a formula which intuitively means that 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.
Formally, we define the semantics of CTL formulae with respect to a structure $M = (S, A_1, \ldots, A_k, L)$ which consists of:

- $S$ - a countable set of states,
- $A_i \subseteq S \times S$, a binary relation on $S$ giving the possible transitions by process $i$, and
- $L$ - 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 $(s_0, s_1, s_2, \ldots) \in S^{\omega}$ 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 $s \xrightarrow{t} t$ is an arc in the tree if $(s, t) \in A_i$.

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:

$$s_0 \models p \quad \text{iff} \quad p \in L(s_0)$$

$$s_0 \models \neg f \quad \text{iff} \quad \text{not} \ (s_0 \models f)$$

$$s_0 \models f_1 \land f_2 \quad \text{iff} \quad s_0 \models f_1 \text{ and } s_0 \models f_2$$

$$s_0 \models \exists x_j f \quad \text{iff for some state } t \text{ such that } (s_0, t) \in A_j, \ t \models f$$

$$s_0 \models A[f_1 \cup f_2] \quad \text{iff for all paths } (s_0, s_1, \ldots), \ \exists i (i \geq 0 \land s_i \models f_2)$$

$$\land \forall j (0 \leq j \land j < i \rightarrow s_j \models f_1)$$

$$s_0 \models E[f_1 \cup f_2] \quad \text{iff for some path } (s_0, s_1, \ldots), \ \exists i (i \geq 0 \land s_i \models f_2)$$

$$\land \forall j (0 \leq j \land j < i \rightarrow s_j \models f_1)$$

We write $\models f$ 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:

- $f_1 \lor f_2 \equiv \neg (\neg f_1 \land \neg f_2)$, $f_1 \land f_2 \equiv \neg (\neg f_1 \lor \neg f_2)$, and $f_1 \iff f_2 \equiv (f_1 \land f_2) \land (f_2 \land f_1)$ for logical disjunction, implication, and equivalence, respectively.

$A[f_1 \cup f_2] \equiv \neg \neg f_1 \cup \neg f_2$ 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 $s$.

$E[f_1 \cup f_2] \equiv \neg A[\neg f_1 \cup \neg f_2]$ 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 $s$.

$\text{AFF}_1 \equiv A[\text{true } U f_1]$ which means for every path, there exists a state on the path at which $f_1$ holds.

$\text{EFF}_1 \equiv E[\text{true } U f_1]$ which means for some path, there exists a state on the path at which $f_1$ holds.

$\text{AGF}_1 \equiv \neg \text{EF } \neg f_1$ which means for every path, at every node on the path $f_1$ holds.

$\text{EGF}_1 \equiv \neg \text{AF } \neg f_1$ which means for some path, at every node on the path $f_1$ holds.
$AX_i f = \neg EX_i \neg f$ which means at all successor states reachable by an atomic step of process $P_i$, $f$ holds.

$EXF = EX_1 f \lor \ldots \lor EX_k f$ which means at some successor state $f$ holds.

$AXf = \neg EX \neg f$ which means at all successor states $f$ holds.

4. MODEL CHECKER

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

$$f \in \text{label}(s) \text{ iff } M, s \models f$$

$$\neg f \in \text{label}(s) \text{ iff } M, s \models \neg f.$$

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 $\neg f$ 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 $A[f_1 U f_2]$ information from the successor states of $s$ (as well as from $s$ itself) is used. Since $A[f_1 U f_2] = f_2 \lor (f_1 \land AXA[f_1 U f_2])$, $A[f_1 U f_2]$ 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 $A[f_1 U f_2]$ is in the set of each immediate successor state of $s$.

Satisfaction of $A[f_1 U f_2]$ may be seen to "radiate" outward from states where it holds immediately by virtue of $f_2$ holding:

Let

$$(A[f_1 U f_2])^0 = f_2$$

$$(A[f_1 U f_2])^{k+1} = f_2 \lor AX(A[f_1 U f_2])^k.$$

It can be shown that $M, s \models (A[f_1 U f_2])^k$ iff $M, s \models A[f_1 U f_2]$ and along every path starting at $s$, $f_2$ holds by the $k$-th state following $s$. Thus, states where $(A[f_1 U f_2])^0$ holds are found first, then states where $(A[f_1 U f_2])^1$ holds, etc. If $A[f_1 U f_2]$ holds, then $(A[f_1 U f_2])^{\text{card}(S)}$ 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, $A[f_1 U f_2]$ has still not been found to hold at state $s$, then put $\neg A[f_1 U f_2]$ in the set for $s$.

The algorithm for pass $i$ is listed below in an Algol-like syntax:
for every state \( s \in S \) do
  for every \( f \in \text{sub}^+(f'_0) \) of length \( i \) do
    if \( f = A[f_1 | f_2] \) and \( f_2 \in \text{set}(s) \) or
    \( f = E[f_1 | f_2] \) and \( f_2 \in \text{set}(s) \) or
    \( f = \text{EX}_{f_1} \) and \( \exists t ((s, t) \in A_j \text{ and } f_1 \in \text{set}(t)) \) or
    \( f = f_1 \wedge f_2 \) and \( f_1 \in \text{set}(s) \) and \( f_2 \in \text{set}(s) \)
    then add \( f \) to set(s)
  end
end;

A: for \( j = 1 \) to \( \text{card}(S) \) do
  for every state \( s \in S \) do
    for every \( f \in \text{sub}^+(f'_0) \) of length \( i \) do
      if \( f = A[f_1 | f_2] \) and \( f_1 \in \text{set}(s) \) and \( \forall t ((s, t) \in A + f \in \text{set}(t)) \) or
      \( f = E[f_1 | f_2] \) and \( f_1 \in \text{set}(s) \) and \( \exists t ((s, t) \in A \wedge f \in \text{set}(t)) \)
      then add \( f \) to set(s)
    end
  end
end;

B: end

c: end

Figures 4.1-4.4 give snapshots of the algorithm in operation on the structure shown for the formula \( AFb \wedge EGa \) (which abbreviates \( AFb \wedge \neg Af-a \)).

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

**Proposition 4.1.** Let \( M = (S, A_1, \ldots, A_k, L) \) be a structure and \( s \in S \). Then \( M, s \models EPp \) if and only if 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.  

**Proof.** (Only if:) Suppose \( M, s \models EPp \). 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 \( i \), and
3. \( M, s' \models p \).

If \( s' \) is a successor of itself, we are done. Otherwise, there is a finite path \( (s', s'', \ldots, s') \) 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 \( s \), \( M, s \models EPp \). 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 at $s'$ which goes through $s'$ infinitely often. Since $s'$ is reachable from $s$, $M, s \models \mathcal{F} p$.

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

$$E[Fg_1 \land \ldots \land Fg_n \land Gh_1 \land \ldots \land Gh_n]$$

meaning

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

sometimes $g_1$ and ... and sometimes $g_n$ and

always $h_1$ and ... and always $h_n".

We claim that the problem of determining whether a given formula $f$ holds in a given finite structure $M$ is NP-hard.

**PROPOSITION 4.2.** 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 $p_1, \ldots, p_n$):

$$E[Fp_1 \land \ldots \land Fp_n \land G(p_1 + XGp_1) \land \ldots \land G(p_n + XGp_n)].$$

**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 vera) and a sink node $u_2$ which is accessible from all $v_i$ (but not vice vera).

Formally, let the structure $M = (U, B, L)$ consist of

$$U = V \cup \{u_1, u_2\} \text{ where } u_1, u_2 \not\in V$$

$L$, on assignment of states to propositions such that

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

$u_1 \not\models p_i$, $u_2 \not\models p_i$ $(1 < i \leq n)$ 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

\[ M, u_1 \vdash f \text{ iff there is a directed infinite path in } M \text{ starting at } u_1 \]
\[ \text{which goes through all } v_i \in V \text{ exactly once and ends in the self-loop through } u_2; \]
\[ \text{iff there is a directed Hamiltonian path in } G. \]

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.

5. THE DECISION PROCEDURE

In this section we outline a tableau-based decision procedure for satisfiability of CTL formulae. Our algorithm is similar to one proposed for UB in \[\text{[BM81].}\]

Tableau-based decision procedures for simpler program logics such as PDL and DPDL are given in \[\text{[PR77] and [BH81].}\] The reader should consult \[\text{[HC68]}\] for a discussion of tableau-based decision procedures for classical modal logics and \[\text{[SM68]}\] for a discussion of tableau-based decision procedures for propositional logic.

We now briefly describe our decision procedure for CTL and illustrate it with a simple example. The decision procedure is described in detail in the full paper. To simplify the notation in the present discussion, 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 satisfiable, 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.
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 \( G_1, G_2, \ldots \) to denote the labels of OR-nodes, \( H_1, H_2, \ldots \) to denote the labels of AND-nodes, and \( F_1, F_2, \ldots \) to denote the labels of arbitrary nodes of either type. No two AND-nodes have the same label, and no two

*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 of \([BM81]\) based on different techniques is forthcoming.
OR-nodes have the same label. The intended meaning is that, when node $F$ is considered as a state in an appropriate structure, $F = f$ 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 $G$, $\text{Blocks}(G) = \{H_1, H_2, \ldots, H_k\}$ has the property that

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

We can explain the construction of $\text{Blocks}(G)$ as follows: Each formula in $G$ may be viewed as a conjunctive formula $\alpha = \alpha_1 \land \alpha_2$ or a disjunctive formula $\beta = \beta_1 \lor \beta_2$. 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, $\exists Fp = p \lor EX\exists Fp$ is a $\beta$ formula and $A\exists p = p \lor AXA\exists p$ 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 $H_1$ of $\text{Blocks}(G)$ by repeatedly expanding each nonelementary formula in $G$ into its $\alpha$ or $\beta$ components. Each $\beta$ expansion yields two blocks, one with $\beta_1$ and one with $\beta_2$. Expansion stops when all $H_i$ are downward closed.

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 $\{A X h_1, A X h_2, E X g_1, E X g_2, E X g_3\}$ is the set of all elementary formulae in $H$, then $\text{Tiles}(H) = \{\{h_1, h_2, g_1\}, \{h_1, h_2, g_2\}, \{h_1, h_2, g_3\}\}$.

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 $\exists Fp \land \exists Fp$ is satisfiable. We build the tableau $T$ starting with root node $G_0 = \{\exists Fp \land \exists Fp\}$. We construct $\text{Blocks}(G_0) = \{H_0, H_1, H_2, H_3\}$. 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 $G_1 = \{\exists Fp\}$ 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, $G_2 \in \text{Tiles}(H_2) \land \text{Tiles}(H_3)$. Continuing in this fashion we obtain the complete tableau shown in Fig. 5.1.
Figure 5.1
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 is possible for eventuality formulae such as $\mathcal{A}Fh$ or $\mathcal{E}Fh$ to be fulfilled: e.g., if $\mathcal{E}Fh \in F$, 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, $EFp \land EF\neg p$ is satisfiable.

Finally, we construct a model $M$ of $EFp \land EF\neg p$. 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 $f \in \mathcal{E}H$. The root of $M$ can be any consistent state $H_1 \in \text{Blocks}(G_0)$. 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.

6. 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. 6.1.

Figure 6.1

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 \( \text{TRY}_i \). From there it enters \( \text{CS}_i \) as soon as possible, provided that the mutual exclusion constraint \( \sim (\text{CS}_1 \land \text{CS}_2) \) is not violated. It remains in \( \text{CS}_i \) as long as necessary to perform its "critical" computations and then re-enters \( \text{NCS}_i \). 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
   \[ \text{NCS}_1 \land \text{NCS}_2 \]

2. mutual exclusion
   \[ \text{AG}(\sim (\text{CS}_1 \land \text{CS}_2)) \]

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

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

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

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
   \[ \text{AG}(\text{CS}_i \rightarrow \text{EX}_i \text{NCS}_i) \]

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

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

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 \( f_1 \land \ldots \land f_n \) to the set of formulae \( \{f_1, \ldots, f_n\} \). (Recall that the set of formulae \( \{f_1, \ldots, f_n\} \) is satisfiable iff \( f_1 \land \ldots \land f_n \) is satisfiable.)

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

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

(4) If we have \( \text{TRY}_1 \) present, there is no need to record \( \sim\text{NCS}_1 \) and \( \sim\text{CS}_1 \). If we have \( \text{NCS}_1 \) present, there is no need to record \( \sim\text{TRY}_1 \) and \( \sim\text{CS}_1 \). If we have \( \text{CS}_1 \) present, there is no need to record \( \sim\text{NCS}_1 \) and \( \sim\text{TRY}_1 \).

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. We then build the tableau using the information about \( \text{Blocks} \) and \( \text{Tiles} \) contained in the list. Next we apply the marking rules to delete inconsistent nodes.

Note that the OR-node \( <\text{CS}_1 \text{ CS}_2 \text{ AFCS}_2> \) is marked as deleted because of a propositional inconsistency (with \( \sim(<\text{CS}_1 \land \text{CS}_2>) \), a consequence of the unwritten invariance \( \text{AG}(\sim(<\text{CS}_1 \text{ CS}_2>)) \). This, in turn, causes the AND-node that is the predecessor of \( <\text{CS}_1 \text{ CS}_2 \text{ AFCS}_2> \) to be marked. The resulting tableau is shown in Fig. 6.2. Each node in Fig. 6.2 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.

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, 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. 6.3). This yields the portion of the model shown in Fig. 6.4.

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 \( (H_0 \rightarrow H_1 \rightarrow H_2 \rightarrow H_0) \). In general, a cycle can be dangerous because it might form a path along which some eventuality is never fulfilled; however, there is no problem this time because the root of a
Figure 6.2
fragment, \( H_1 \), occurs along the cycle. A fragment root serves as a checkpoint to ensure that all eventualities are fulfilled. By symmetry between the roles of 1 and 2, we add in the arcs \( H_8 \perp H_{10} \) and \( H_8 \parallel H_0 \). The structure now has the form shown in Fig. 6.5.

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 \( H_9 \perp H_4 \). Again a cycle is formed but since \( H_4 \) is a fragment root no problems arise. Similarly, we add in the arc \( H_{10} \parallel H_1 \). The decision procedure thus yields a model \( M \) such that

\[ M_{s_0} \models f_0 \]

where \( f_0 \) is the conjunction of the mutual exclusion system specifications.

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 distinguished by their propositional labels. In order to distinguish \( H_6 \) from \( H_7 \), we introduce a variable \( \text{TURN} \) which is set to 1 upon entry to \( H_6 \) and to 2 upon entry to \( H_7 \). If we introduce \( \text{TURN}'s \) value into the labels of \( H_6 \) and \( H_7 \) then, the labels uniquely identify each node in the global system flowgraph. See Fig. 6.6.

We describe how to obtain the synchronization skeletons of the individual processes from the global system flowgraph. In the sequel we will refer to these global system states by the propositional labels.

When \( P_1 \) is in \( \text{NCS}_1 \), there are three possible global states \( [\text{NCS}_1 \text{ NCS}_2], [\text{NCS}_1 \text{ TRY}_2], \) and \( [\text{NCS}_1 \text{ CS}_2] \). In each case it is always possible for \( P_1 \) to make a transition into \( \text{TRY}_1 \) by the global transitions \( [\text{NCS}_1 \text{ NCS}_2] \perp [\text{TRY}_1 \text{ NCS}_2], [\text{NCS}_1 \text{ TRY}_2] \perp [\text{TRY}_1 \text{ TRY}_2], \) and \( [\text{NCS}_1 \text{ CS}_2] \perp [\text{TRY}_1 \text{ CS}_2] \). 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.

Now when \( P_1 \) is in \( \text{TRY}_1 \), there are four possible global states: \( [\text{TRY}_1 \text{ NCS}_2], [\text{TRY}_1 \text{ TRY}_2 \text{ TURN}=1], [\text{TRY}_1 \text{ TRY}_2 \text{ TURN}=2], \) and \( [\text{TRY}_1 \text{ CS}_2] \) and their associated global transitions by \( P_1 \): \( [\text{TRY}_1 \text{ NCS}_2] \perp [\text{CS}_1 \text{ NCS}_2] \) and \( [\text{TRY}_1 \text{ TRY}_2 \text{ TURN}=1] \perp [\text{CS}_1 \text{ TRY}_2] \). (No transitions by \( P_1 \) are possible in \( [\text{TRY}_1 \text{ TRY}_2 \text{ TURN}=2] \) or \( [\text{TRY}_1 \text{ CS}_2] \).) When \( P_1 \) is in \( \text{CS}_1 \) the associated global states and transitions are: \( [\text{CS}_1 \text{ NCS}_2], [\text{CS}_1 \text{ TRY}_2], [\text{CS}_1 \text{ NCS}_2] \perp [\text{NCS}_1 \text{ NCS}_2], \) and \( [\text{CS}_1 \text{ TRY}_2] \perp [\text{NCS}_1 \text{ TRY}_2] \). Altogether, the synchronization skeleton for \( P_1 \) is shown in Fig. 6.7(a). By symmetry in the global state diagram we obtain the synchronization skeleton for \( P_2 \) as shown in Fig. 6.7(b).

The general method of factoring out the synchronization skeletons of the individual processes may be described as follows: Take the model of the specification formula and retain only the propositional formulae in the labels of each node.
Figure 6.7 (a)

Figure 6.7 (b)
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 $v_1,...,v_n$, then for each $v_i$, set $L:=i$ on all arcs coming into $v_i$ and add $L:=1$ 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 $P_j$ which has $m$ distinct code regions $R_1,...,R_m$. Initially, the synchronization skeleton for $P_j$ is a graph with $m$ distinct nodes $R_1,...,R_m$ 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

$$v((S_1 \land ... \land S_p) : [R_j \ldots S_p] \downarrow [R_k \ldots S_p] \text{ is an arc in the global system flowgraph}).$$

Add $L:=n$ to the label of $R_j \rightarrow R_k$ if some arc $[R_j S_1 \ldots S_p] \xrightarrow{L:=n} [R_k S_1 \ldots S_p]$ also occurs in the flowgraph.

7. 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 will 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.
8. BIBLIOGRAPHY


