USING BRANCHING TIME TEMPORAL LOGIC TO SYNTHESIZE SYNCHRONIZATION SKELETONS[1]

E. Allen Emerson and Edmund M. Clarke[2]

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-208 September 1982
Revised version

[1]This work was partially supported by NSF Grant MCS-7908365.

[2]Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213.

Science of Computer Programming 2 (1982) 241–266
North-Holland

USING BRANCHING TIME TEMPORAL LOGIC TO SYNTHESIZE SYNCHRONIZATION SKELETONS*

E. Allen EMERSON
Computer Sciences Department, University of Texas at Austin, Austin, TX 78712, U.S.A.

Edmund M. CLARKE
Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A.

Communicated by K. Apt
Received November 1981
Revised September 1982

Astract. We present a method of constructing concurrent programs in which the *synchronization skeleton* of the program is automatically synthesized from a (branching time) temporal logic specification. The synthesis method uses a decision procedure based on the finite model property of the logic to determine satisfiability of the specification formula f. If f is satisfiable, then 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.

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). We describe a decision procedure which, given a formula

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

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.

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. The decision procedure is described in Section 4. Section 5 then shows how the synthesis method can be used to construct solutions to common concurrent programming problems such as the starvation-free mutual exclusion problem and the readers–writers problem. Finally, Section 6 compares our work to related efforts, and Section 7 presents some concluding remarks.

2. Model of parallel computation

We consider nonterminating concurrent programs of the form $P = P_1 \| \cdots \| P_n$ which consist of a finite number of fixed sequential processes $P_1, \ldots, P_n$ running in parallel. We observe that for most actual concurrent programs the portions of each process responsible for interprocess synchronization can be cleanly separated from the sequential applications-oriented computations performed by the process. This suggests that we focus our attention on *synchronization skeletons* which are abstractions of actual concurrent programs where detail irrelevant to synchronization is suppressed.

We may view the synchronization skeleton of an individual process P_i as a flowgraph where each node represents a region of code intended to perform some sequential computation and each arc represents a conditional transition (between different regions of sequential code) used to enforce synchronization constraints. For example, there may be a node labelled CS_i representing 'the critical section of process P_i''. While in CS_i, the process P_i may simply increment a single variable x, or it may perform an extensive series of updates on a large database. In general, the internal structure and intended application of the regions of sequential code in an actual concurrent program are unspecified in the synchronization skeleton. The only assumptions we make about the sequential computation performed in such a region of code by an actual program corresponding to the synchronization skeleton are that

(i) it always terminates, and

(ii) the set of variables it accesses is disjoint from the set of variables used for synchronization.

Under these assumptions, we can eliminate all steps of the sequential computation from consideration.

Formally, the synchronization skeleton of each process P_i is a directed graph where each node is labelled by a unique name, and each arc is labelled with a synchronization command $B? \rightarrow A$ consisting of an enabling condition (i.e., guard) B and corresponding action A to be performed. (Self-loops, where there is an arc from a node to itself, are disallowed.) A synchronization *state* is a tuple of the form $(N_1, \ldots, N_n, x_1, \ldots, x_m)$ where each N_i is the current node of P_i and $x_1, \ldots, x_m$ is a list (possibly empty) of auxiliary synchronization variables. A guard B is a predicate on states and an action A is a function which updates the values of the auxiliary variables. If the guard B is omitted from a command, it is interpreted as *true* and we simply write the command as A. If the action A is omitted, the auxiliary variables are unaltered and we write the command as $B?$.

We model parallelism in the usual way by the nondeterministic interleaving of the 'atomic' transitions of the individual synchronization skeletons of the processes P_i. Hence, at each step of the computation, some process with an enabled transition is nondeterministically selected to be executed next. Assume that the current state is $(N_1, \ldots, N_i, \ldots, N_n, x_1, \ldots, x_m)$ and that process P_i contains an arc from node N_i to N_i' labelled by the command $B? \rightarrow A$. If B is true in the current state then a permissible next state is $(N_1, \ldots, N_i', \ldots, N_n, x_1', \ldots, x_m')$ where $x_1', \ldots, x_m'$ is the list of updated auxiliary variables resulting from the action A. A computation path is any infinite sequence of states where each successive pair of states is related by the above next state relation. (Since we are concerned with nonterminating processes, we, in general, assume that some process is always enabled.)

The behavior of a program 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 transition 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 program beginning in the given start state. Temporal logic specifications may then be thought of as making statements about patterns of behavior in the computation trees. The synthesis task thus amounts to supplying the commands to label the arcs of each process's synchronization skeleton so that the resulting computation trees of the entire program $(P_1 \| \cdots \| P_k)$ meet a given temporal logic specification.

Finally, we note the following points about our model:

(1) Since all components of a state are accessible to each process, synchronization is, in effect, accomplished through shared memory with test-and-set primitives;

(2) The synchronization skeletons that we synthesize will be correct under the assumption of pure nondeterministic scheduling. They will also be correct under fair scheduling assumptions, but fairness is a stronger condition than we need.

The reader may wish to compare our model with that of Pnueli [17].

3. The specification language

Our specification language is a (propositional) branching time temporal logic which we call "Computation Tree Logic" (CTL). It is related to the logic of "Unified Branching Time" (UB) discussed in [3] and to the language of "Computation Tree Formulae" (CTF) proposed in [8].

We have the following syntax for CTL (where p denotes an atomic proposition, and f and g denote (sub-) formulae):

(1) Each of p, $f \wedge g$, and $\sim f$ is a formula (where the latter two constructs indicate conjunction and negation, respectively);

(2) $EX_j f$ 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 holds;

(3) $A[fUg]$ is a formula which intuitively means that for every computation path, there is some state along the path where g holds, and f holds at every state along the path *until* g;

(4) $E[fUg]$ is a formula which intuitively means that for some computation path, there is some state along the path where g holds, and f holds at every state along the path *until* g.

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 – a labelling of each state with the set of atomic propositions true in the state.

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

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

$s_0 \models p$ iff $p \in L(s_0)$,
$s_0 \models \sim f$ iff not $(s_0 \models f)$,
$s_0 \models f \wedge g$ iff $s_0 \models f$ and $s_0 \models g$,
$s_0 \models EX_j f$ iff for some state t, $(s_0, t) \in A_j$ and $t \models f$,
$s_0 \models A[fUg]$ iff for all fullpaths $(s_0, s_1, \ldots)$,
$\exists i [i \geq 0 \wedge s_i \models g \wedge \forall j (0 \leq j \wedge j < i \Rightarrow s_j \models f)]$,
$s_0 \models E[fUg]$ iff for some fullpath $(s_0, s_1, \ldots)$,
$\exists i [i \geq 0 \wedge s_i \models g \wedge \forall j (0 \leq j \wedge j < i \Rightarrow s_j \models f)]$.

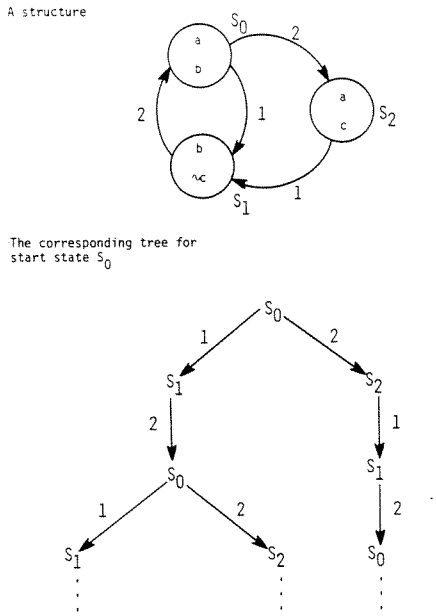


Fig. 1.

We write $\models f$ to indicate that f is *valid*, i.e., true at all states in all structures. Similarly, we write $\exists f$ to indicate that f is *satisfiable*, i.e., true in some states of some structure.

We introduce the abbreviations $f \vee g$ for $\sim(\sim f \wedge \sim g)$, $f \Rightarrow g$ for $\sim f \vee g$, and $f \equiv g$ for $(f \Rightarrow g) \wedge (g \Rightarrow f)$ indicating logical disjunction, implication, and equivalence, respectively. We also introduce a number of additional modalities as abbreviations: $A[fWg]$ for $\sim E[\sim fU \sim g]$, $E[fWg]$ for $\sim A[\sim fU \sim g]$, AFf for $A[true\ Uf]$, EFf for $E[true\ Uf]$, AGf for $\sim EF \sim f$, EGf for $\sim AF \sim f$, $AX_i f$ for $\sim EX_i \sim f$, EXf for $EX_1 f_1 \vee \cdots \vee EX_k f_k$, and AXf for $AX_1 f \wedge \cdots \wedge AX_k f$. Particularly useful modalities are AFf, which means that for every path, there exists a state on the path where f holds, and AGf, which means that f holds at every state along every path.

A formula of the form $A[fUg]$ or $E[fUg]$ is an *eventuality* formula. An eventuality corresponds to a liveness property in that it makes a promise that something does happen. This promise must be *fulfilled*. The eventuality $A[fUg](E[fUg])$ is fulfilled for s in M provided that for every (respectively, for some) path starting at s, there exists a finite prefix of the path in M whose last state satisfies g and all of whose other states satisfy f. Since AFg and EFg are special cases of $A[fUg]$ and $E[fUg]$, respectively, they are also eventualities. In contrast, $A[fWg]$, $E[fWg]$ (and their special cases AGg and EGg) are *invariance* formulae. An invariance corresponds to a safety property since it asserts that whatever happens to occur (if anything) will meet certain conditions.

4. The decision procedure

In this section we describe a tableau-based decision procedure for satisfiability of CTL formulae. Our algorithm is similar to one proposed for UB in [3].[1] Tableau-based decision procedures for simpler program logics such as PDL and DPDL are given in [18] and [2]. The reader should consult [12] for a discussion of tableau-based decision procedures for classical modal logics and [20] for a discussion of tableau-based decision procedures for propositional logic.

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 $D_1, D_2, \ldots$ to denote the labels of OR-nodes, $C_1, C_2, \ldots$ to denote the labels of AND-nodes, and $B_1, B_2, \ldots$ 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 B is considered as a state in an appropriate structure, $B \models f$ for all $f \in B$. The tableau T has a *root* node $D_0 = \{f_0\}$ from which all other nodes in T are accessible.

The set of successors of each OR-node D, $Blocks(D) = \{C_1, C_2, \ldots, C_k\}$, has the property that

$$\models D \text{ iff } \models C_1 \text{ or } \ldots \text{ or } \models C_k.$$

Similarly, the set of successors of each AND-node C, $Tiles(C) = \{D_1, D_2, \ldots, D_k\}$, has the property that, if C contains no propositional inconsistencies, then

$$\models C \text{ iff } \models D_1 \text{ and } \cdots \text{ and } \models D_k.$$

The following subsections describe the decision procedure in greater detail. (Section 5 illustrates the use of the decision procedure in program synthesis.)

4.1. *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 $D_0 = \{f_0\}$.

[1] The [3] algorithm is incorrect and will claim that certain satisfiable formulae are unsatisfiable. Ben-Ari [1] states that a corrected version, using different techniques, is forthcoming. A proof of correctness for a tableau-based procedure for UB similar to the one described here is given in [7]. Also, a filtration-based decision procedure and an alternative tableau-based decision procedure for the uniprocessor version of CTL (which subsumes UB) are given in [9] along with proofs of their correctness.

(2) If all nodes in T have successors, halt. Otherwise, let B be any node without successors in T. If B is an OR-node D, construct $Blocks(D) = \{C_1, \ldots, C_k\}$ and attach each C_i as an immediate successor of D in T. If any C_i has the same label as another AND-node C already present in T, then merge C_i and C. If B is an AND-node C, construct $Tiles(C) = \{D_1, \ldots, D_k\}$ and attach each D_i as an immediate successor of D in T. Label the arc (C, D_i) in T with each j such that $D_i \in Tiles_j(C)$. If any D_i has the same label as some other OR-node D already present in T, then merge D_i and D. Repeat this step.

4.2. Construction of Blocks(D)

For convenience, we assume that every formula in D has been placed in *standard* form with all negations driven inside so that only atomic propositions appear negated. (This can be done using duality: $\sim(f \wedge g) \equiv \sim f \vee \sim g$, $\sim AFh \equiv EG \sim h$, etc.) We say that a formula is *elementary* provided that it is a proposition, the negation of a proposition, or has main connective AX_j or EX_j. Any other formula is *nonelementary*. Each nonelementary formula in D may be viewed as a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$. Clearly, $f \wedge g$ is an α formula and $f \vee g$ is a β formula. A modal formula may be classified as α or β based on its fixpoint characterization (cf. [8]); thus, $EFg \equiv g \vee EXEFg$ is a β formula and $AGg \equiv g \wedge AXAGg$ is an α formula. The following table summarizes the classification:

$\alpha = f \wedge g$	$\alpha_1 = f$	$\alpha_2 = g$
$\alpha = A[fWg]$	$\alpha_1 = g$	$\alpha_2 = f \vee AXA[fWg]$
$\alpha = E[fWg]$	$\alpha_1 = g$	$\alpha_2 = f \vee EXE[fWg]$
$\alpha = AGg$	$\alpha_1 = g$	$\alpha_2 = AXAGg$
$\alpha = EGg$	$\alpha_1 = g$	$\alpha_2 = EXEGg$
$\beta = f \vee g$	$\beta_1 = f$	$\beta_2 = g$
$\beta = A[fUg]$	$\beta_1 = g$	$\beta_2 = f \wedge AXA[fUg]$
$\beta = E[fUg]$	$\beta_1 = g$	$\beta_2 = f \wedge EXE[fUg]$
$\beta = AFg$	$\beta_1 = g$	$\beta_2 = AXAFg$
$\beta = EFg$	$\beta_1 = g$	$\beta_2 = EXEFg$

To construct $Blocks(D)$ 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 [20].) Initially, let the root $= D$. In general, let B be a leaf in the tree constructed so far for which there exists a nonelementary formula $f \in B$. If $f = \alpha$, add a single son to B with the label $B \setminus \{\alpha\} \cup \{\alpha_1, \alpha_2\}$. If $f = \beta$, add two sons to B, one labelled $B \setminus \{\beta\} \cup \{\beta_1\}$ and the other labelled $B \setminus \{\beta\} \cup \{\beta_2\}$. Eventually, this construction must halt because all leaves $B_1, \ldots, B_m$ will contain only elementary formulae. (This can be proved by induction of the length of the longest formula in D.) Then let $Blocks(D) = \{C_1, \ldots, C_m\}$ where C_i is the set of all formulae appearing in some node on the path from B_i back to the root of the tree.

4.3. Construction of Tiles(C).

For each $j \in [1{:}k]$, we must first determine the set $Tiles_j(C)$ of successors associated with process j.[2] Let

$$CA_j = \{f: AX_j f \in C\} \quad \text{and} \quad CE_j = \{g: EX_j g \in C\}.$$

If $CE_j \neq \emptyset$ then write CE_j as $\{g_1, \ldots, g_n\}$ and define

$$Tiles_j(C) = \{D_j^1, \ldots, D_j^n\} \quad \text{where } D_j^i = CA_j \cup \{g_i\} \text{ for } i \in [1{:}m].$$

If $CE_j = \emptyset$ then let $Tiles_j(C) = \emptyset$. Now define the set of all successors of C,

$$Tiles(C) = \bigcup\{Tiles_j(C): j \in [1{:}k]\}.$$

If $D_i \in Tiles(C)$ then the arc from C to D_i in T is labelled with $j_1, \ldots, j_m$ where $D_i \in Tiles_{j_1}(C), \ldots, Tiles_{j_m}(C)$. Fig. 2 gives an example.

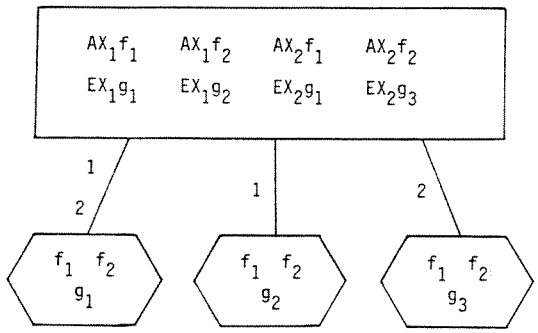


Fig. 2.

There are two special cases to consider. Let $CA = \bigcup\{CA_j: j \in [1{:}k]\}$ and $CE = \bigcup\{CE_j: j \in [1{:}k]\}$. Note that if CE is empty then $Tiles(C)$ is also empty, whereas each node in the tableau should have a successor to properly reflect the requirement that each state in a structure has a successor. If both CA and CE are empty then we simply add a 'dummy' successor to C: let $Tiles(C) = \{D\}$ where $D = \{f: f \in C\}$ and let $Blocks(D) = \{C\}$. If only CE is empty, then split C into $C_1, \ldots, C_k$ where each $C_j = C \cup \{EX_j True\}$ and recompute $Tiles(C_j)$ for each j separately.

4.4 Deleting inconsistent portions of the tableau

We now apply the rules below to delete as inconsistent certain nodes of the tableau T. First we need the following technical definition:

A *full subdag* Q rooted at node B in T is a finite, directed acyclic subgraph of T satisfying the following three conditions:

(1) For every OR-node $D \in Q$, there exists precisely one AND-node C such that C is a son of D in Q and in T,

[2] We use the notation $[m{:}n]$ to indicate $\{x: x \text{ is a natural number and } m \leq x \leq n\}$.

(2) For every AND-node $C \in Q$, if C has any sons at all in Q, then every son of C in T is a son of C in Q;

(3) B is the unique node in Q from which all other nodes are reachable.

Note that a full subdag Q 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 Q (although they may very well have successors when considered as nodes in T). All nodes of the frontier are AND-nodes.

Here are the deletion rules:

DeleteP: Delete any node B which is immediately inconsistent, i.e., contains a formula f and its negation $\sim f$.

DeleteOR: Delete any OR-node D all of whose original AND-node sons C_i are already deleted.

DeleteAND: Delete any AND-node C one of whose original OR-node sons D_j has already been deleted.

DeleteEU: Delete any node B such that $E[fUg] \in B$ and there does *not* exist some AND-node C' reachable from B such that $g \in C'$ and for all AND-nodes C'' on some path from C' back to B, $f \in C''$.

DeleteAU: Delete any node B such that $A[fUg] \in B$ and there does *not* exist a full subdag Q rooted at B such that for all nodes C' on the frontier of Q, $g \in C'$ and for all non-frontier AND-nodes C'' in Q, $f \in C''$.

DeleteEF: Delete any node B such that $EFg \in B$ and there does *not* exist some AND-node C' reachable from B such that $g \in C'$.

DeleteAF: Delete any node B such that $AFg \in B$ and there does *not* exist a full subdag Q rooted at B such that for all nodes C' on the frontier of Q, $g \in C'$.

Apply the deletion rules as long as possible. Each time a node is deleted, delete all incident arcs as well. Deletion must eventually stop because each successful application of a deletion rule deletes one node and there are only a finite number of nodes in T.

If the root of T is deleted, then f is unsatisfiable. If the root of T is undeleted, then the subgraph of T induced by the remaining undeleted nodes can be unraveled into a finite model of f_0.

4.5. Unravelling the tableau into a model

Let T^* be the subgraph of T that remains after all nodes have been deleted using the rules above. We will construct a finite model M of f_0 by 'unravelling' T^*: For each AND-node C in T^*, and for each eventuality formula $g \in C$, there is a subdag, $DAG[C, g]$, rooted at C which certifies that g is fulfilled. (We know this subdag exists because C is not marked by one of the rules for AF, EF, AU, or EU on account of g.) We use these subdags to construct, for each AND-node C, a model fragment $FRAG[C]$ such that every eventuality in C is fulfilled within $FRAG[C]$. We then splice together these fragments to obtain M (cf. [2, 9]).

4.6. Selecting subdags

If C is in T^* and $g \in C$ is an eventuality formula, then there is a subdag rooted at C 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.

We first consider the case where $g = A[fUh]$. Initially, we set $tag(C) = 0$ for all AND-nodes C such that $h \in C$ and we set $tag(B) = \infty$ for all other nodes B. Then we let the size of subdags radiate outward by making $card(T^*)$ passes over the tableau. During each pass we perform the following step for each node B:

if B is an AND-node C such that $A[fUh] \in C$ and $tag(C) = \infty$ and $tag(D) < \infty$ for all $D \in Tiles(C)$ and $f \in C$
then let $tag(C) := 1 + \max\{tag(D): D \in Tiles(C)\}$;
if B is an OR-node D such that $A[fUh] \in D$ and $tag(D) = \infty$ and $tag(C) < \infty$ for some $C \in Blocks(D)$
then let $tag(D) := \min\{tag(C): C \in Blocks(D)\}$;

After executing all $card(T^*)$ passes, if, for AND-node C, $tag(C) = k < \infty$ then there will be a full subdag for g rooted at C of minimal size $= 2k$. To select a specific full subdag Q we perform a construction in stages.

Initially let Q_0 consist of the single node C.

In general, obtain Q_{i+1} from Q_i as follows:

for all nodes $B \in frontier(Q_i)$ **do**
  if B = some OR-node D
  then choose an AND-node $C \in Blocks(D)$ with a minimal tag value
    (if there is more than one C eligible, choose one with a
    maximal $card(Tiles(C))$ value;
    if there is still more than one C eligible, choose the
    one of lowest index in a predefined ordering.)
    attach C as the successor of D;
  if B = some AND-node C
  then add each member of $Tiles(C)$ as a successor of B

Halt with $Q = Q_i$ when all frontier nodes of Q_i are AND-nodes C' with $tag(C') = 0$. Let $DAG[C, g]$ denote the subdag naturally induced by the AND-nodes of D. (Note: $g = AFh$ is a special case of $A[fUh]$ where $f = true$.)

The construction when $g = E[fUh]$ is similar. Let $tag(C) = 0$ for all AND-nodes such that $h \in C$ and set $tag(B) = \infty$ for all other nodes B. Then make $card(T^*)$ passes over T^* performing the following step for each node B:

if B is an AND-node C such that $E[fUh] \in C$ and $tag(C) = \infty$ and $tag(D) < \infty$ for some $D \in Tiles(C)$ and $f \in C$
then let $tag(C) := 1 + \min\{tag(D): D \in Tiles(C)\}$;

if B is an OR-node D such that $E[fUh] \in D$ and $tag(D) = \infty$ and
$tag(C) < \infty$ for some $C \in Blocks(D)$
then let $tag(D) := \min\{tag(C): C \in Blocks(D)\}$;

After performing this tagging procedure, if, for AND-node C, $tag(C) = k < \infty$ then there is a path of length $2k$ from C to AND-node C' such that $h \in C'$ and $f \in C''$ for each AND-node C'' on the path up to but not including C'. We can then trace out a minimal length path $C = B_0, B_1, \ldots, B_{2k+1} = C'$. Start with $B_0 = C$ and, in general, choose B_{i+1} to be a successor of B_i of minimal tag value. This path has the form $C_0, D_0, C_1, D_1, \cdots, D_{k-1}, C_k$. Form the path of AND-nodes $C = C_0, C_1, \ldots, C_k = C'$. For each C_i and for each $D \in Tiles(C_i)$, choose a $C' \in Blocks(D)$ and attach a copy of it as a successor of C_i. The resulting graph is $DAG[C_0, g]$ which can be used in building the model of f_0. (Note: $g = EFh$ is a special case of $E[fUh]$ where $f = true$.)

4.7. Construction of fragments from dags

For each AND-node C in T^*, we construct the fragment $FRAG[C]$ to have these properties:

(1) $FRAG[C]$ is a dag with root C consisting of (copies of) AND-nodes.

(2) $FRAG[C]$ is *generated* by T^* in this sense: for all nodes C_0 in $FRAG[C]$ if $\{C_1, \ldots, C_m\}$ is the set of successors of C_0 in $FRAG[C]$, then there exist OR-nodes $D_1, \ldots, D_m$ in T^* such that $Tiles(C_0) = \{D_1, \ldots, D_m\}$ and $C_i \in Blocks(D_i)$ for all $i \in [1{:}m]$. If the arc (C_0, C_i) in $FRAG[C]$ has labels $j_1, \ldots, j_n$ then the arc (C_0, D_i) has labels $j_1, \ldots, j_n$ in T^*.

(3) All eventuality formulae in C are fulfilled for C in $FRAG[C]$.

We construct $FRAG[C]$ in stages. Let $g_1, g_2, \ldots, g_m$ be a list of all eventuality formulae occurring in C. We build a sequence of dags $FRAG_1, \ldots, FRAG_m = FRAG[C]$ so that, for each $j \in [1{:}m]$, $FRAG_j$ is a subgraph of $FRAG_{j+1}$ and $g_1, \ldots, g_j$ are fulfilled for C in $FRAG_j$.

Let $FRAG_1 = DAG[C, g_1]$. To obtain $FRAG_{i+1}$ from $FRAG_i$ do the following:

for *all* $C' \in frontier(FRAG_i)$ **do**
  if $g_{i+1} \in C'$
  then attach (a copy of) $DAG[C', g_{i+1}]$ to $FRAG_i$ at C'
end

Finally, let $FRAG[C] = FRAG_m$.

4.8. Constructing the model from fragments

We construct M by splicing together fragments. Again, the construction is done in stages:

Let $M_1 = C_0$ where $C_0 \in Blocks(\{f_0\})$ is chosen as in step [2.2]. To construct M_{k+1} from M_k perform the following procedure:

[1] If $frontier(M_k) \neq \emptyset$, choose an arbitrary frontier node C of M_k; otherwise, halt.

[2] For each $D_i \in Tiles(C)$ do the steps below:

[2.1] If there is some $C_j \in Blocks(D_i)$ such that C_j occurs in M_k and every cycle that would result from adding the arc (C, C_j) contains a fragment root, then do add (C, C_j) to M_k and continue with the next D_i. Otherwise, do step [2.2].

[2.2] Choose C' to be some $C_j \in Blocks(D_i)$ such that $FRAG[C_j]$ is of minimal size. (Choose one with a maximal number of successors among those C_j with fragments of minimal size, and break ties by choosing the one with lowest index in a predefined ordering.[3]) Attach $FRAG[C']$ to M_k by the arc (C, C'). Continue with the next D_i.

Note: $D_i \in Tiles_{j_1}(C), \ldots, Tiles_{j_m}(C)$ for some $j_1, \ldots, j_m$. Any arc added in [2.1] or [2.2] is labelled with $j_1, \ldots, j_m$.

[3] Call the resulting graph M_{k+1}. Repeat step [1].

The construction halts with $k = N$ when $frontier(M_k)$ is empty. Let $M = M_N$.

5. The synthesis method

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 demonstrate the synthesis method on an instance of the starvation-free mutual exclusion problem, a version of the readers–writers problem, and an inconsistent problem specification.

5.1. Mutual exclusion problem

We first 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 *N*on*C*ritical *S*ection
TRY_i	the *TRY*ing Section
CS_i	the *C*ritical *S*ection

which it moves through as suggested in Fig. 3.

[3] We choose a node of maximal outdegree to increase the degree of nondeterministic choice in an effort to maximize potential parallelism.

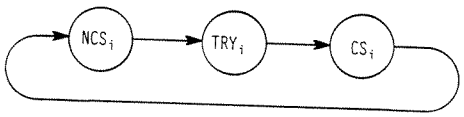


Fig. 3.

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 $\sim(CS_1 \wedge CS_2)$ is not violated. It remains in CS_i as long as necessary to perform its 'critical' computations and then re-enters 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

$$NCS_1 \wedge NCS_2.$$

(2) mutual exclusion

$$AG(\sim(CS_1 \wedge CS_2)).$$

(3) absence of starvation for P_i

$$AG(TRY_i \Rightarrow AFCS_i).$$

(4) each process P_i is always in exactly one of the three code regions

$$AG(NCS_i \vee TRY_i \vee CS_i),$$
$$AG(NCS_i \Rightarrow \sim(TRY_i \vee CS_i)),$$
$$AG(TRY_i \Rightarrow \sim(NCS_i \vee CS_i)),$$
$$AG(CS_i \Rightarrow \sim(NCS_i \vee TRY_i)).$$

(5) it is always the case that any move P_i makes from its noncritical region is into its trying region and such a move is always possible

$$AG(NCS_i \Rightarrow (AX_iTRY_i \wedge EX_iTRY_i)).$$

(6) it is always the case that any move P_i makes from its trying region is into its critical region

$$AG(TRY_i \Rightarrow AX_iCS_i).$$

(7) it is always the case that any move P_i makes from its critical region is into its noncritical region and such a move is always possible

$$AG(CS_i \Rightarrow (AX_iNCS_i \wedge EX_iNCS_i)).$$

(8) a transition by one process cannot cause a move by the other

$$AG(NCS_i \Rightarrow AX_j NCS_i),$$
$$AG(TRY_i \Rightarrow AX_j TRY_i),$$
$$AG(CS_i \Rightarrow AX_j CS_i).$$

(9) some process can always move

$$AG(EX\,True).$$

(*Note*: In the above specifications $i, j \in [1{:}2]$ and $i \neq j$.)

Remark. Specifications 4–9 describe what may be thought of as the local structure of the synchronization skeletons. They formally specify the information informally communicated by Fig. 3. In contrast, specifications 1–3 describe the global behavior of the system and constitute what we ordinarily (and inaccurately) think of as 'the problem specification'. All the information in specifications 1–9 is needed to give a precise problem description from which a solution can be synthesized. However, once the local structure specifications are set up, complete specifications of new problems can be obtained by simply varying the global behavior assertions. For instance, we obtain our second and third examples by altering specification 3.

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

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

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

(4) If we have TRY_i present, there is no need to record $\sim NCS_i$ and $\sim CS_i$. If we have NCS_i present, there is no need to record $\sim TRY_i$ and $\sim CS_i$. If we have CS_i present, there is no need to record $\sim NCS_i$ and $\sim TRY_i$.

By the above conventions, the root node of the tableau will have the two formulae NCS_1 and NCS_2 recorded in its label which we now write as $\langle NCS_1\, NCS_2\rangle$. In building the tableau, it will be helpful to have constructed $Blocks(D)$ for the following OR-nodes: $\langle NCS_1\, NCS_2\rangle$, $\langle TRY_1\, NCS_2\rangle$, $\langle CS_1\, NCS_2\rangle$, $\langle TRY_1\, TRY_2\rangle$, and $\langle CS_1\, TRY_2\rangle$. For all other OR-nodes D' appearing in the tableau, $Blocks(D')$ will be identical to or can be obtained by symmetry from $Blocks(D)$ for some D in the above list. Figures 4–8 show the abbreviated construction of $Blocks(D)$ for these

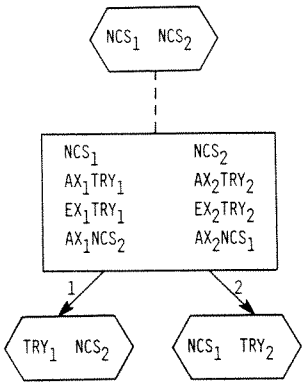


Fig. 4.

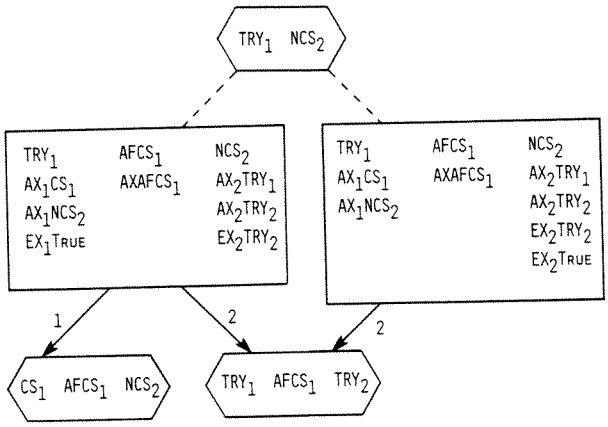


Fig. 5.

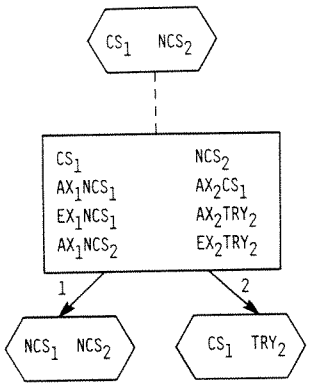


Fig. 6.

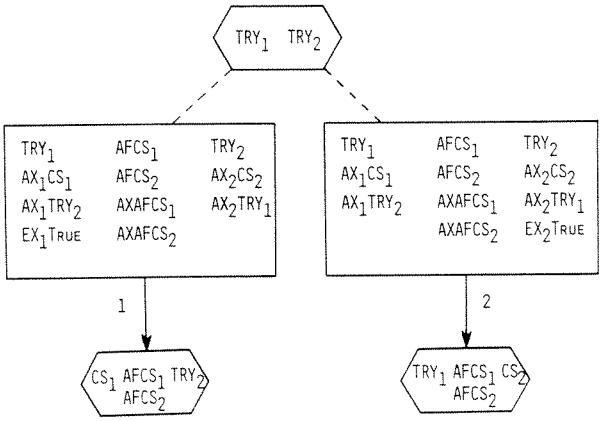


Fig. 7.

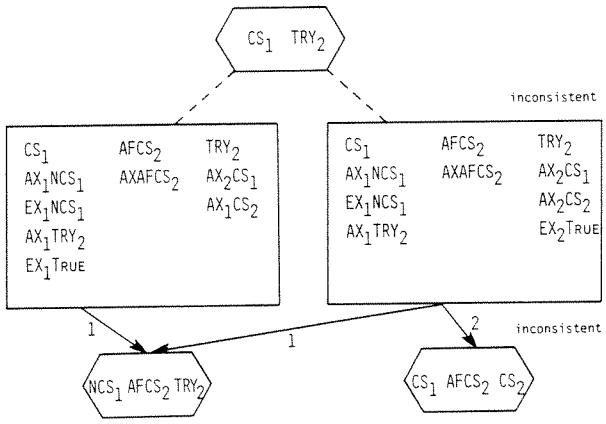


Fig. 8.

OR-nodes as well as *Tiles*(C) for each $C \in$ *Blocks*(D). We then build the tableau using the information about *Blocks* and *Tiles* contained in Figs. 4–8. We next apply the deletion rules to detect inconsistent nodes. Note that the OR-node $\langle CS_1\, CS_2\, AFCS_2\rangle$ is deleted because of a propositional inconsistency with $\sim(CS_1 \wedge CS_2)$, a consequence of the unwritten invariance $AG(\sim(CS_1 \wedge CS_2))$. This, in turn, causes the AND-node that is the predecessor of $\langle CS_1\, CS_2\, AFCS_2\rangle$ to be deleted. The resulting tableau is shown in Fig. 9 where each node 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. As explained in Section 4, 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.

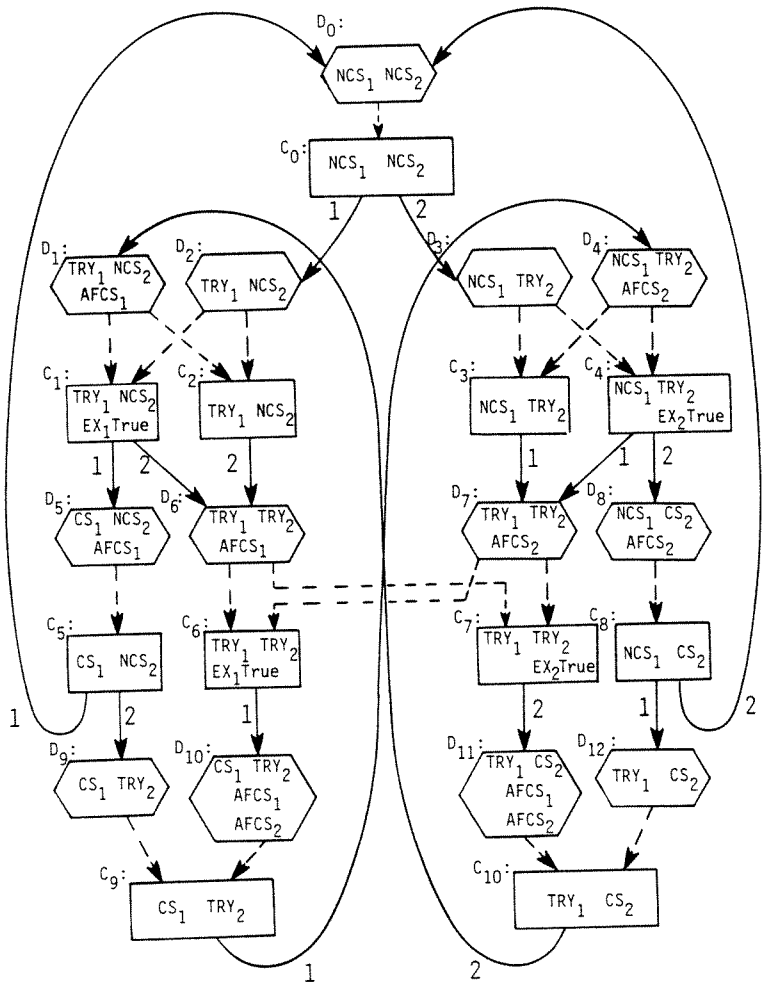


Fig. 9. Tableau for mutual exclusion problem.

The root node of the model is C_0, the unique successor of D_0. From the tableau we see that C_0 must have two successors, one of C_1 or C_2 and one of C_3 or C_4. Each candidate successor state contains an eventuality to fulfill, so we must construct and attach its fragment. Using the method described in Section 4, we choose the fragment rooted at C_1 to be the left successor and the fragment rooted at C_4 to be the right successor. This yields the portion of the model contained within contour (a) in Fig. 10.

We continue the construction by finding successors for each of the leaves: C_5, C_9, C_{10}, and C_8. We start with C_5. By inspection of T, we see that the only successors C_5 can have are C_0 and C_9. Since C_0 and C_9 already occur in the structure built so far, we add the arcs $C_5 \xrightarrow{1} C_0$ and $C_5 \xrightarrow{2} C_9$ to the structure. Note that this introduces a cycle $(C_0 \xrightarrow{1} C_1 \xrightarrow{1} C_5 \xrightarrow{1} C_0)$. In general, a cycle can be dangerous because it might

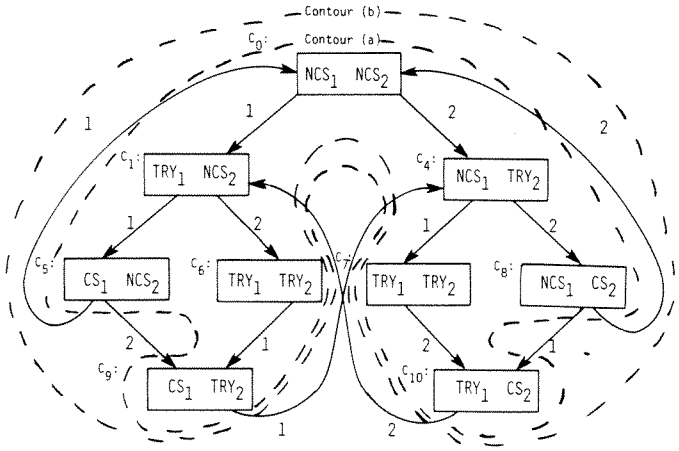


Fig. 10. Construction of model for mutual exclusion problem.

form a path along which some eventuality is never fulfilled; however, there is no problem this time because the root of a fragment, C_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 $C_8 \xrightarrow{1} C_{10}$ and $C_8 \xrightarrow{2} C_0$. The structure now has the form suggested by contour (b) in Fig. 10.

We now have two leaves remaining: C_9 and C_{10}. We see from the tableau that C_4 is a possible successor to C_9. We add in the arc $C_9 \xrightarrow{1} C_4$. Again a cycle is formed but since C_4 is a fragment root no problems arise. Similarly, we add in the arc $C_{10} \xrightarrow{2} C_1$. 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. The entire model is shown in Fig. 10 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 C_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 C_1. Note that all states except C_6 and C_7 are distinguished by their propositional labels. In order to distinguish C_6 from C_7, we introduce an auxiliary variable *TURN* which is set to 1 upon entry to C_6 and to 2 upon entry to C_7. If we introduce *TURN*'s value into the labels of C_6 and C_7, then the labels uniquely identify each node in the global system flowgraph. See Fig. 11.

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 NCS_1, there are three possible global states: $[NCS_1\, NCS_2]$, $[NCS_1\, TRY_2]$, $[NCS_1\, CS_2]$. In each case it is always possible for P_1 to make a transition into TRY_1 by the global transitions $[NCS_1\, NCS_2] \xrightarrow{1} [TRY_1\, NCS_2]$,

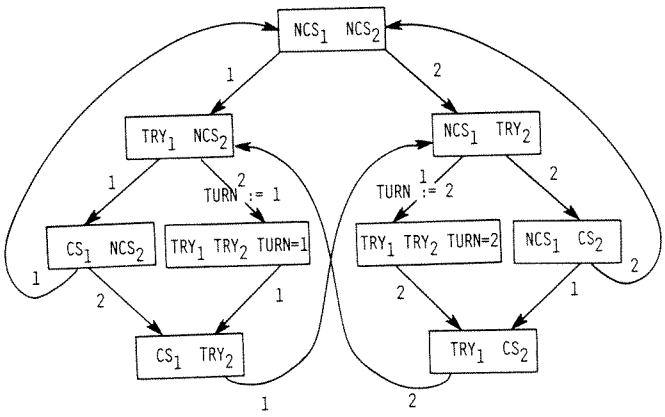
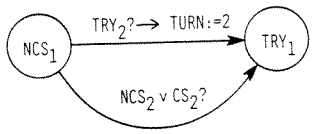
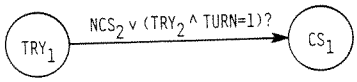


Fig. 11. Global system flowgraph for mutual exclusion problem.

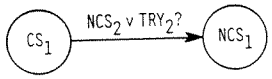
$[NCS_1\ TRY_2] \xrightarrow{1,TURN:=2} [TRY_1\ TRY_2]$, and $[NCS_1\ CS_2] \xrightarrow{1} [TRY_1 CS_2]$. From each global transition by P_1, we obtain a transition in the synchronization skeleton of P_1. The P_2 component of the global state provides enabling conditions for the transitions in the skeleton of P_1. If along a global transition, there is an assignment to *TURN*, the assignment is copied into the action of the corresponding transition of the synchronization skeleton. We merge the transitions which lack assignments to obtain the portion of the synchronization skeleton of P_1 shown in Fig. 12(a).



a



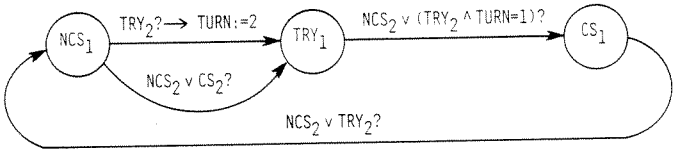
b



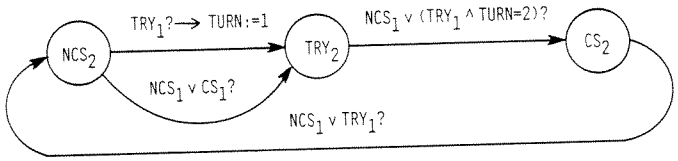
c

Fig. 12.

Now when P_1 is in TRY_1, there are four possible global states: $[TRY_1\ NCS_2]$, $[TRY_1\ TRY_2\ TURN = 1]$, $[TRY_1\ TRY_2\ TURN = 2]$, and $[TRY_1\ CS_2]$ and their associated global transitions by P_1: $[TRY_1\ NCS_2] \xrightarrow{1} [CS_1\ NCS_2]$ and $[TRY_1\ TRY_2\ TURN = 1] \xrightarrow{1} [CS_1\ TRY_2]$. (No transitions by P_1 are possible in $[TRY_1\ TRY_2\ TURN = 2]$ or $[TRY_1\ CS_2]$.) Thus we obtain the portion of the synchronization skeleton for P_1 shown in Fig. 12(b). When P_1 is in CS_1 the associated global states and transitions are: $[CS_1\ NCS_2]$, $[CS_1\ TRY_2]$, $[CS_1\ NCS_2] \xrightarrow{1} [NCS_1\ NCS_2]$, and $[CS_1\ TRY_2] \xrightarrow{1} [NCS_1\ TRY_2]$ from which we obtain the portion of the synchronization skeleton for P_1 shown in Fig. 12(c). Altogether, the synchronization skeleton for P_1 is shown in Fig. 13(a). By symmetry in the global state diagram we obtain the synchronization skeleton for P_2 as shown in Fig. 13(b).



a



b

Fig. 13. (a) Synchronization skeleton for P_1 of mutual exclusion problem; (b) Synchronization skeleton for P_2 of mutual exclusion problem.

5.2. *Readers–writers problem*

We now solve a simplified version of the readers–writers problem with writer priority. Let P_1 be the reader process and P_2 the writer process. Then, to obtain a specification of the new problem, replace number 3 in the specification of the starvation-free mutual exclusion problem with the following formulae:

(3a) absence of starvation for P_1 provided P_2 remains in its noncritical region

$$AG(TRY_1 \Rightarrow AF(CS_1 \vee \sim NCS_2)).$$

(3b) absence of starvation for P_2

$$AG(TRY_2 \Rightarrow AF\, CS_2).$$

(3c) priority of P_2 over P_1 for outstanding requests to enter the critical region

$$AG((TRY_1 \wedge TRY_2) \Rightarrow A[TRY_1\ U\ CS_2]).$$

The resulting set of CTL formulae specifies the readers–writers system. (Note: if formulae (3a) were $AG(TRY_1 \Rightarrow AFCS_1)$, the set of formulae would be unsatisfiable. This is demonstrated in the next example.)

The new specifications (3a), (3b), and (3c) have no significant effect upon *Blocks*(D) for most OR-nodes D. However, *Blocks*(D) changes substantially for the OR-node $\langle TRY_1\ TRY_2\rangle$ and Fig. 14 shows its abbreviated construction. We

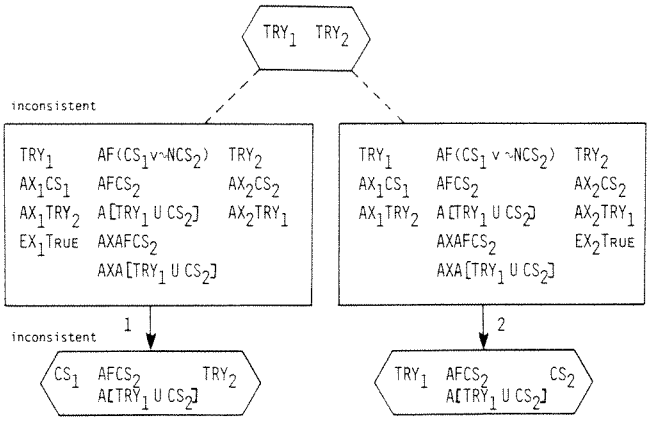


Fig. 14.

then build the tableau as before using the information about *Blocks* and *Tiles* in the figures for the above OR-nodes. The resulting tableau T is shown in Fig. 15 where each node is labelled with a minimal set of formulae sufficient to distinguish it from any other node.

We construct a model M from T using the same method that was used for the mutual exclusion problem. The model is shown in Fig. 16 where only the propositions true in a state are retained in the label. Since all states are distinguished by their propositional labels, there is no need to introduce auxiliary variables, and the synchronization skeletons of the individual processes may be extracted immediately. The synchronization skeleton for P_1 is shown in Fig. 17(a) and for P_2 in Fig. 17(b).

5.3. An inconsistent problem specification

Finally, we give an example that illustrates the ability of the synthesis algorithm to detect inconsistent (i.e., unsatisfiable) specifications. Suppose that we formulate the readers–writers problem using the formula (3a′) shown below instead of (3a):

(3a′) $AG(TRY_1 \Rightarrow AFCS_1)$.

This results in essentially the same tableau as before, except the stronger eventuality $AFCS_1$ replaces the weaker eventuality $AF(CS_1 \vee \sim NCS_2)$. However, the new tableau is inconsistent because $AFCS_1$ cannot be fulfilled: When we apply the deletion rules, we will not be able to find a full subdag certifying fulfillment of

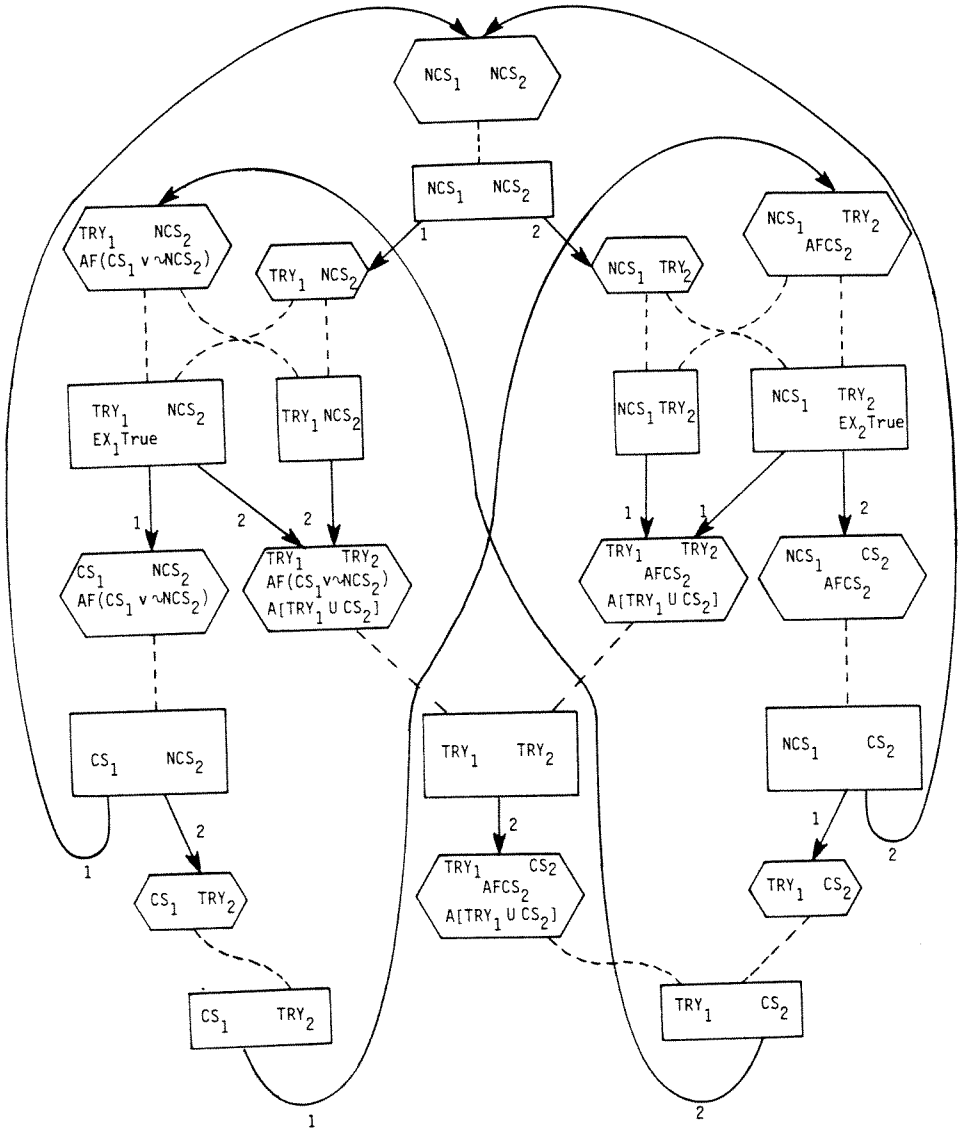


Fig. 15. Tableau for readers–writers problem.

$AFCS_1$ rooted at any node that does not itself already contain CS_1. Nodes such as the AND-node $[TRY_1\ TRY_2]$ will be marked inconsistent and these inconsistencies will be propagated up to the root of the tableau. Thus, the set of specification formulae is unsatisfiable. (Note: This formalizes our intuition that it is impossible for both processes to be assured of inevitably entering their critical regions while giving P_2 priority over P_1: if P_2 runs fast enough, it can continually outpace P_1. For example, in the global flowgraph for the satisfiable version of the readers–writers problem shown in Fig. 16, the system can cycle endlessly through the following

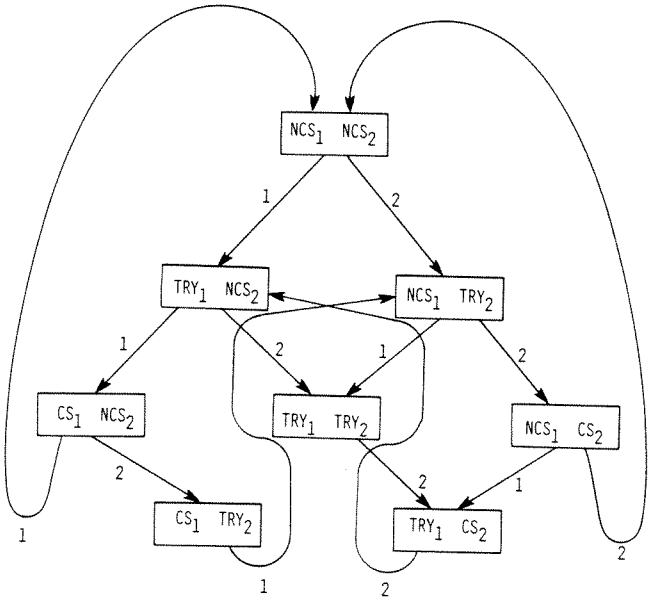


Fig. 16. The model for the readers–writers problem.

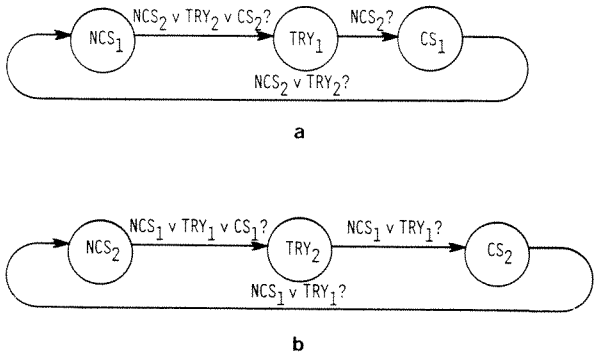


Fig. 17. (a) Synchronization skeleton of reader P_1; (b) Synchronization skeleton for writer P_2.

sequence of transitions:

$$[TRY_1\ TRY_2] \xrightarrow{2} [TRY_1\ CS_2] \xrightarrow{2} [TRY_1\ NCS_2] \xrightarrow{2} [TRY_1\ TRY_2].$$

Such a situation is, in general, unavoidable.)

5.4. Factoring out synchronization skeletons

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.

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, let $x_L := i$ on all arcs coming into v_i and add $x_L = i$ 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 node 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 $L_j \xrightarrow{i} L_k$ 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 of P_i and is labelled with a command having the enabling condition

$$\bigvee\{(S_1 \wedge \cdots \wedge S_p) : [R_j S_1 \cdots S_p] \xrightarrow{i} [R_k S_1 \cdots S_p]$$

is an arc in the global system flowgraph$\}$.

Add $x_L := n$ to the action in the command labelling $R_j \rightarrow R_k$ whenever some arc $[R_j S_1 \cdots S_p] \xrightarrow{i, x_L := n} [R_k S_1 \ldots S_p]$ also occurs in the flowgraph.

6. Related work

There have been other efforts toward parallel program synthesis. In particular, Manna and Wolper [15, 23] have independently developed model-theoretic synthesis techniques similar to ours. Both our method and theirs revolve around the same central concept: to synthesize a concurrent program from a temporal logic description of its intended behavior by applying a decision procedure to the specification formula and then extracting the individual processes from the finite model that results (assuming that the formula is satisfiable). However, the methods differ substantially in their orientation and in the technical machinery used to realize the concept:

(1) Manna and Wolper synthesize CSP programs. Their model of parallel computation is thus based on message passing primitives in a distributed computing environment whereas ours is oriented toward test-and-set primitives in a shared memory environment. Note, however, that their approach also involves some degree of centralization since all interprocess communication occurs between a distinguished synchronizer process and one of its satellite processes.

(2) The particular temporal logic systems used have incomparable expressive power. For example, using the techniques of [23] it is possible to synthesize a program such that, along all computation paths, a condition holds at all *even* time steps. The logic we use cannot express this particular property. Conversely, certain properties are expressible in our logic but not in theirs (see below).

(3) Manna and Wolper use a linear time logic for specification whereas we use a branching time logic (cf. [13]). We prefer a branching time logic because it enables us to assert directly in the logic the *existence* of computation paths having specified properties. This can be helpful in ensuring that the synthesized program exhibits an adequate degree of parallelism (i.e., that the synthesized program can follow any one of a number of computation paths and is not a 'degenerate' solution with only a single path). In branching time logic we can write $AF(P \vee Q) \wedge EFP \wedge EFQ$ to ensure that

(i) along every path either P or Q occurs,
(ii) there is at least one path where P occurs, and
(iii) there is at least one path where Q occurs.

However, no system of linear time logic allows us to naturally assert the existence of alternative paths. For example, the linear time specification $FP \vee FQ$ is met by a program that also meets the specification FP and has no computation path where Q occurs. On the other hand, linear time logic provides greater simplicity and many people feel it easier to use.

Earlier approaches to parallel program synthesis can be found in the work of Laventhal [14] and Ramamritham and Keller [19]. Laventhal uses a specification language that is essentially predicate calculus augmented with a special predicate to define the relative order of events in time. Ramamritham and Keller use an applied linear time temporal logic. Instead of model-theoretic methods, both [14] and [19] use *ad hoc* techniques to construct a monitor that meets the specification.

It is also possible to use model-theoretic temporal logic techniques to automatically verify the correctness of certain *a priori* existing concurrent programs. Clarke, Emerson, and Sistla [4, 5] describe an efficient algorithm (a *model checker*) to decide whether a given finite structure is a model of a particular formula. Since the global system flowgraph of a finite-state concurrent system may be viewed as defining a finite structure, the model checker can be used to mechanically verify the correctness of finite-state concurrent programs.

7. Conclusion

We have shown that it is possible to automatically synthesize the synchronization skeleton of a concurrent program from a temporal logic specification. Can such a synthesis method be developed into a practical software tool? Recall that while deciding satisfiability of propositional calculus formulae requires exponential time in the worst case using the best known algorithms, the average case performance is substantially better and working automatic theorem provers and program verifiers are a reality. Similarly, the average case performance of the decision procedure used by the synthesis method may be substantially better than the potentially exponential time worst case. Furthermore, synchronization skeletons are generally small. We therefore believe that this approach may in the long run turn out to be quite practical. We encourage additional research in this area.

References

[1] M. Ben-Ari, Personal Communication (1981).

[2] M. Ben-Ari, J. Halpern and A. Pnueli, Finite models for deterministic propositional dynamic logic, *Proc. 8th International Colloquium on Automata, Languages, and Programming* (1981) 249–263.

[3] M. Ben-Ari, Z. Manna and A. Pnueli, The temporal logic of branching time, *Proc. 8th Annual ACM Symposium on Principles of Programming Languages* (1981) 164–176.

[4] E.M. Clarke and E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic (Abridged version), *Proc. IBM Workshop on Logics of Programs*, Lecture Notes in Computer Science **131** (Springer, Berlin, 1981) 52–71; The full version appears as Aiken Computation Lab TR-12-81, Harvard University (1981).

[5] E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic verification of finite state concurrent systems: A practical approach, to be presented at the *10th Annual ACM Symposium on Principles of Programming Languages* (1983).

[6] E.M. Clarke, Program invariants as fixpoints, *Computing* **21** (4) 273–294.

[7] E.A. Emerson, Branching time temporal logic and the design of correct concurrent programs, Ph.D. Thesis, Division of Applied Sciences, Harvard University (1981).

[8] E.A. Emerson and E.M. Clarke, Characterizing correctness properties of parallel programs as fixpoints, *Proc. 7th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science **85** (Springer, Berlin, 1980) 169–181.

[9] E.A. Emerson and J.Y. Halpern, Decision procedures and expressiveness in the temporal logic of branching time, *Proc. 14th Annual ACM Symposium on Theory of Computing* (1982) 169–180.

[10] L. Flon and N. Suzuki, The total correctness of parallel programs, *SIAM J. Comput.* (May 1981) 227–246.

[11] D. Gabbay, A. Pnueli et al., The temporal analysis of fairness, *Proc. 7th Annual ACM Symposium on Principles of Programming Languages* (1980) 163–173.

[12] G. Hughes and M. Cresswell, *An Introduction to Modal Logic* (Methuen, London, 1968).

[13] L. Lamport, 'Sometimes' is sometimes 'not never', *Proc. 7th Annual ACM Symposium on Principles of Programming Languages* (1980) 174–185.

[14] M. Laventhal, Synthesis of synchronization code for data abstractions, Ph.D. Thesis, M.I.T. (1978).

[15] Z. Manna and P. Wolper, Synthesis of communicating processes from temporal logic specifications, *Proc. IBM Workshop on Logics of Programs*, Lecture Notes in Computer Science **131** (Springer, Berlin, 1981) 253–281.

[16] S. Owicki and L. Lamport, Proving liveness properties of concurrent programs, *ACM Trans. Programming Languages* **4** (3) 455–495.

[17] A. Pnueli, The temporal semantics of concurrent programs, *Theoret. Comput. Sci.* **13** (1981) 45–60.

[18] V. Pratt, A near optimal method for reasoning about action, *J. Comput. Systems Sci.* **20** (1980) 231–254.

[19] K. Ramamritham and R. Keller, Specification and synthesis of synchronizers, *Proc. 9th International Conference on Parallel Processing* (1980) 311–321.

[20] R. M. Smullyan, *First Order Logic* (Springer, Berlin, 1968).

[21] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific J. Math.* **5** (1955) 285–309.

[22] P. Wolper, Temporal logic can be more expressive, *Proc. 22nd Annual Symposium on Foundations of Computer Science* (1981) 340–347.

[23] P. Wolper, Specification and synthesis of communicating processes using an extended temporal logic (Preliminary version), *Proc. 9th Annual ACM Symposium on Principles of Programming Languages* (1982) 20–33.