Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation

PAUL C. ATTIE
Northeastern University and MIT Laboratory for Computer Science
and
E. ALLEN EMERSON
The University of Texas at Austin

Methods for mechanically synthesizing concurrent programs from temporal logic specifications have been proposed by Emerson and Clarke and by Manna and Wolper. An important advantage of these synthesis methods is that they obviate the need to manually compose a program and manually construct a proof of its correctness. A serious drawback of these methods in practice, however, is that they produce concurrent programs for models of computation that are often unrealistic, involving highly centralized system architecture (Manna and Wolper), processes with global information about the system state (Emerson and Clarke), or reactive modules that can read all of their inputs in one atomic step (Anuchitanukul and Manna, and Pnueli and Rosner). Even simple synchronization protocols based on atomic read/write primitives such as Peterson's solution to the mutual exclusion problem have remained outside the scope of practical mechanical synthesis methods. In this paper, we show how to mechanically synthesize in more realistic computational models solutions to synchronization problems. We illustrate the method by synthesizing Peterson's solution to the mutual exclusion problem.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.2 [**Programming Techniques**]: Automatic Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/ Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*state diagrams*; D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods, model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*logics of programs, mechanical verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*temporal logic*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms: Design, Languages, Theory, Verification

Additional Key Words and Phrases: Atomic registers, concurrent programs, program synthesis, specification, temporal logic

An extended abstract containing some of these results was presented at the ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, 1996, under the title "Synthesis of Concurrent Systems for an Atomic Read / Atomic Write Model of Computation." P.C. Attie was supported in part by the National Science Foundation under CAREER grant number CCR-0096356. E. A. Emerson was supported in part by NSF Grants CCR-9804736 and CCR-0098141 together with Texas ARP project 003658-0650-1999.

Authors' addresses: P.C. Attie, College of Computer Science, Northeastern University, Boston, MA 02115; E.A. Emerson, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

1. INTRODUCTION

Methods for synthesizing (finite state) concurrent programs from temporal logic specifications based on the use of a decision procedure for testing temporal satisfiability have been proposed by Emerson and Clarke [1982] and Manna and Wolper [1984]. An important advantage of these synthesis methods is that they obviate the need to manually compose a program and manually construct a proof of its correctness. One only has to formulate a precise problem specification; the synthesis method then mechanically constructs a correct solution. A serious drawback of these methods in practice, however, is that they produce concurrent programs for restricted models of parallel computation. For example, the method of Manna and Wolper [1984] produces CSP programs; in other words, programs with synchronous message passing. Moreover all communication takes place between a central synchronizing process and one of its satellite processes, and thus the overall architecture of such programs is highly centralized. The synthesis method of Emerson and Clarke [1982] produces concurrent programs for the shared-memory model of computation. Transitions of such programs are test-and-set operations in which a large number of shared variables can be tested and set in a single transition; in other words, the grain of atomicity is large. More recent synthesis methods [Anuchitanukul and Manna 1994; Pnueli and Rosner 1989a; 1989b] for synthesizing reactive modules assume that all of the inputs to a system can be consolidated into a single input variable [Pnueli and Rosner 1989a; 1989b], or into a global state that the module can read in one step [Anuchitanukul and Manna 1994]. In practice, the primitives provided by hardware are of small (and fixed) granularity, e.g., atomic read and write operations on a single register, or test-and-set operations on a single bit. It is therefore desirable to synthesize concurrent programs with small-grain operations.

In this paper, we present a method for synthesizing concurrent programs for a shared-memory model of computation in which the only operations are atomic reads or atomic writes of single variables. Our method accepts as input a formal specification, which can be expressed either as a finite-state machine, together with a temporal logic formula that specfies additional required properties (safety and liveness), or can be expressed just as a temporal logic formula. In the later case, the formula would be somewhat larger, since it must also express some "frame" properties such as the local structure of each process. From the specification, we first synthesize a correct program that, in general, contains test-and-set and multiple assignment operations. We then decompose these operations into sequences of atomic reads/writes. Finally, we modify the resulting program to ensure that it still satisfies the original specification, since new behaviors (that violate the specification) may have been introduced by the decomposition. We illustrate our method by synthesizing an atomic read/write solution to the mutual exclusion problem.

The paper is organized as follows. Section 2 presents our model of concurrent computation and our specification language. It also reviews the synthesis method of Emerson and Clarke [1982] and provides all the needed material from that earlier work. In particular, it discusses the relationship between programs and the global-state transition diagrams that they give rise to (or, going in the other direction, can be "extracted" from). Section 3 formalizes our notion of concurrent programs in which the only operations are atomic reads or atomic writes of single variables. It also explores the relationship between these atomic read/write programs and the global-state transition diagrams from which such programs can be extracted. This lays the foundation for Section 4, which presents our synthesis method. Section 5 presents an example of the use of our method to synthesize an atomic read/write solution to the two-process mutual exclusion problem. Section 6 extends our method so that it produces programs for an atomic registers model. Section 7 analyzes the space complexity of our method. It then discusses related work, further directions for research, and concludes. Some of the longer proofs are omitted from the main body, and are provided in Appendix A; Appendix B provides a glossary of symbols.

2. PRELIMINARIES

2.1 Model of Concurrent Computation

We consider finite-state concurrent programs of the form $P = P_1 \| \cdots \| P_K$ that consist of a finite number of fixed sequential processes $P_1, \ldots, P_K$ running in parallel. With every process P_i, we associate a single, unique index, namely i. 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 state-machine where each state 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 labeled C_i representing the critical section of process P_i. While in C_i, the process P_i may simply increment a single variable, 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. By virtue of the abstraction to synchronization skeletons, we thus 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 labeled by a unique name (s_i) which represents a *local state* of P_i, and each arc is labeled with a synchronization command $B \rightarrow A$ consisting of an enabling condition (i.e., guard) B and corresponding action A to be performed (i.e., a guarded command [Dijkstra 1976]). Each node must have at least one outgoing arc, i.e., a skeleton contains no "dead ends." A *global state* is a tuple of the form $(s_1, \ldots, s_K, v_1, \ldots, v_m)$ where each node s_i is the current local state of P_i and $v_1, \ldots, v_m$ is a list giving the current values of shared variables $x_1, \ldots, x_m$ (we assume that these are ordered in some fixed way, so that $v_1, \ldots, v_m$ specifies a unique value for each shared variable). A guard B is a predicate on states, and an action A is a parallel assignment statement that updates the values of the shared 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 shared variables are unaltered, and we write the command as B.

We model concurrency 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" arc is nondeterministically selected to be executed next. Assume that the current state is $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ and that process P_i contains an arc from node s_i to node s'_i labeled by the command $B \to A$. If B is true in the current state then a permissible next state is $(s_1, \ldots, s'_i, \ldots, s_K, v'_1, \ldots, v'_m)$ where $v'_1, \ldots, v'_m$ is the list of updated values for the shared variables resulting from the execution of action A. The arc from s_i to s'_i is said to be *enabled* in state s. An arc that is not enabled is *disabled*, or *blocked*. A *computation path* is any sequence of states where each successive pair of states is related by the above next-state relation. Note that the operations involved in just checking the value of a guard B are not explicitly represented by the next-state relation given above. In particular, if B is checked and found to be false, then no change occurs, not even an indication that some computation has occurred at a "lower" implementation level. Essentially, when P_i is in some local state s_i, the guards of all the arcs leaving s_i must be repeatedly checked until one of them is found to be *true*, at which point P_i can execute the corresponding arc. Thus, guards really hide "busy wait" loops (or "await" primitives) where their value is checked. We discuss these issues further in Section 6.

The synthesis task thus amounts to supplying the commands to label the arcs of each process' synchronization skeleton so that the resulting global-state transition diagram of the entire program $P_1 \| \cdots \| P_K$ meets a given temporal logic specification.

2.2 Programs and Global-State Transition Diagrams

We consider the semantics of a program P to be given by the global state transition diagram M that is generated by all the computation paths of P starting from a specified set of initial states S_0. We now formalize our notions of program and global-state transition diagram.

Definition 2.2.1 (*Program*). A *program* $P = P_1 \| \cdots \| P_K$ is the parallel composition of K processes $P_1, \ldots, P_K$. Associated with each process P_i is a set of *atomic propositions* $\mathcal{AP}_i$ (with $\mathcal{AP}_i \cap \mathcal{AP}_j = \emptyset$ when $i \neq j$). Let $\mathcal{AP} = \bigcup_{i \in [1:K]} \mathcal{AP}_i$.[1] Each process P_i is a synchronization skeleton. An arc of P_i (henceforth called a *P_i-arc*) is a tuple $(s_i, B \to A, t_i)$, where s_i, t_i are local states of P_i (henceforth called *P_i-states*), and $B \to A$ is an arc label, consisting of a *P_i-guard* B and a multiple assignment statement A. B is built up from the standard propositional logic connectives, the atomic propositions in $\mathcal{AP} - \mathcal{AP}_i$, and expressions of the form $x = c$ ($c \in D_x$), where x is a shared variable in P, and c is some value drawn from D_x, the domain of x. A multiple assignment statement has the form $//_{m \in [1:n]}\ x^m := c^m$, where x^m is a shared variable in P, and $c^m \in D_{x^m}$, for all $m \in [1:n]$.[2] Associated

[1] We use $[1 : K]$ for the set of natural numbers 1 through K inclusive.

[2] $//$ is the simultaneous parallel assignment operator. Execution of $//_{m \in [1:n]}\ x^m := c^m$ is achieved by executing all of the assignments $x^m := c^m$ simultaneously. Here x^m is a shared variable in P, and $c^m \in D_{x^m}$. Note that the possibility of interference does not arise, since the c^m are all constants. If $n = 0$, then the multiple assignment $//_{m \in [1:n]}\ x^m := c^m$ is written as *skip*.

with each P_i-state s_i is a *local atomic proposition valuation* $\mathcal{L}[s_i] \subseteq \mathcal{AP}_i$. $\mathcal{L}[s_i]$ contains the atomic propositions that are true in s_i. Also, all of the structures defined here are finite, since our programs are finite-state.

We shall use the term "local state" when we intend to say P_i-state for some unspecified process index i. Let $\mathcal{SH}$ denote the set of all shared variables in P, and let $\mathcal{A}$ denote the set of all multiple assignment statements in P.

We shall find it technically convenient in the sequel to "group" the atomic propositions in $\mathcal{AP}_i$ into a single variable L_i whose value in s_i is $\mathcal{L}[s_i]$. We also extend the definition of i-state to provide for a value assigned to L_i by every i-state. L_i may be regarded as a concrete implementation of the atomic propositions in $\mathcal{AP}_i$. For a particular i-state s_i, the corresponding value of L_i is given by

$$s_i(L_i) = \mathcal{L}[s_i], \qquad \text{(LOC)}$$

i.e., L_i is the set of atomic propositions in $\mathcal{AP}_i$ that are true in i-state s_i. In practice, L_i could be encoded efficiently as a bit string. Since the atomic propositions of P_i (those in $\mathcal{AP}_i$) can be read by other processes, and since the value of the atomic propositions of P_i gives some information about the possible current local state of P_i, we shall refer to L_i as the *externally visible location counter* of P_i. However, it is possible for different local states of P_i to have the same local atomic proposition valuation, and thus the same value of L_i. Hence L_i does not uniquely determine the current local state of P_i. Thus, the actual location counter of P_i is properly thought of as having two components: the externally visible L_i, and an internal component (not readable by processes other than P_i) that distinguishes between local states of P_i that happen to have the same local atomic proposition valuation. This internal component is formalized in Section 3.2 below. The presence of this internal component allows P_i to change its local state without executing a write operation to a variable that other processes can read (namely L_i). If every transition of P_i required a write operation to L_i, then P_i could never read the value of a shared variable (or the L_j of some other process P_j) without violating the requirements of the atomic read/write model. Thus, the decomposition of the location counter into an externally visible component and an internal component facilitates the introduction of "read only" transitions during the refinement of the initial high-atomicity program into an atomic read/write program.

If $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ is a global state, then we define $s{\uparrow}i = s_i$ and $s{\uparrow}\mathcal{SH} = (v_1, \ldots, v_m)$.[3] We write $s_i(Q_i), s(Q_i)$ for the value of atomic proposition Q_i in local state s_i, global state s respectively. $s_i(Q_i) = \mathit{true}$ if $Q_i \in \mathcal{L}[s_i]$, and false otherwise. $s(Q_i) \overset{\text{df}}{=} s{\uparrow}i(Q_i)$.

Definition 2.2.2 (*Global-State Transition Diagram*). The *global-state transition diagram* of program $P = P_1 \| \cdots \| P_K$ is a tuple (S_0, S, R) where

(1) S is the set of all global states of $P_1 \| \cdots \| P_K$, and
(2) $S_0 \subseteq S$ is the set of initial states of $P_1 \| \cdots \| P_K$, and

[3] Alternatively, we can consider $s{\uparrow}\mathcal{SH}$ as a set of variable bindings $\{< x_1, v_1 >, \ldots, < x_m, v_m >\}$. Global state s is then the tuple $(s_1, \ldots, s_K, < x_1, v_1 >, \ldots, < x_m, v_m >)$. This viewpoint is useful later in the description of the synthesis method.

(3) $R \subseteq S \times [1:K] \times \mathcal{A} \times S$ is the set of transitions of $P_1 \| \cdots \| P_K$, that is, the set of all transitions $s \xrightarrow{i,A} t$ such that
   (a) $i \in [1:K]$, and
   (b) $s \in S$, $t \in S$ (i.e., s and t are global states of $P_1 \| \cdots \| P_K$), and
   (c) $\wedge j \in [1:K] - \{i\} : s{\uparrow}j = t{\uparrow}j$, and
   (d) there exists an arc $(s{\uparrow}i, B \rightarrow A, t{\uparrow}i)$ in P_i such that
      i. $s(B) = \mathit{true}$,[4] and
      ii. $\langle s{\uparrow}\mathcal{SH}\rangle\, A\, \langle t{\uparrow}\mathcal{SH}\rangle$.

$\langle s{\uparrow}\mathcal{SH}\rangle\, A\, \langle t{\uparrow}\mathcal{SH}\rangle$ is Hoare triple notation [Hoare 1969] for total correctness, which in this case means that execution of A always terminates,[5] and, when the shared variables in $\mathcal{SH}$ have the values assigned by s, leaves these variables with the values assigned by t. Note that whenever we say "state" we intend "global state," unless the context makes it clear that a local state is intended. A transition by P_i (i.e., of the form $s \xrightarrow{i,A} t$) is referred to as a P_i-transition. A (computation) path is therefore a sequence of transitions such that the end state of each transition is the start state of the next transition. An *initialized path* is a path that starts in an initial state. A state is *reachable* iff it lies on some initialized path. A transition is reachable iff its source state is reachable. If all transitions in a global-state transition diagram are reachable, then we say that the global-state transition diagram is in *reachable form.*

Figure 1 gives an example program $P = P_1 \| P_2$ together with the global-state transition diagram M generated by applying Definition 2.2.2 to P. The atomic propositions are $\{D_1, E_1\}$ for process P_1, and $\{D_2, E_2\}$ for process P_2. The initial-state set is $\{s^0\}$. In each global (local) state, we display the propositions that are true.

2.3 The Temporal Logic CTL

CTL [Emerson and Clarke 1982; Emerson 1990] is a propositional branching time temporal logic. We have the following syntax for CTL, where Q_i denotes an atomic proposition in $\mathcal{AP}_i, i \in [1:K]$, and f, g denote (sub-)formulae.

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

(2) $\mathsf{EX}_j f$ is a formula which means that there is an immediate successor state reachable by executing one arc of process $P_j, j \in [1:K]$, in which formula f holds.

(3) $\mathsf{A}[f \mathsf{U} g]$ is a formula which means that for every maximal computation path there is some state along the path where g holds, and f holds at every state along the path until that state.

(4) $\mathsf{E}[f \mathsf{U} g]$ is a formula which means that for some maximal computation path there is some state along the path where g holds, and f holds at every state along the path until that state.

[4] $s(B)$ is defined by the usual inductive scheme: $s(\text{“}x = c\text{”}) = \mathit{true}$ iff $s(x) = c$, $s(B1 \wedge B2) = \mathit{true}$ iff $s(B1) = \mathit{true}$ and $s(B2) = \mathit{true}$, $s(\neg B1) = \mathit{true}$ iff $s(B1) = \mathit{false}$.

[5] Termination is obvious, since the right-hand side of A is a list of constants.

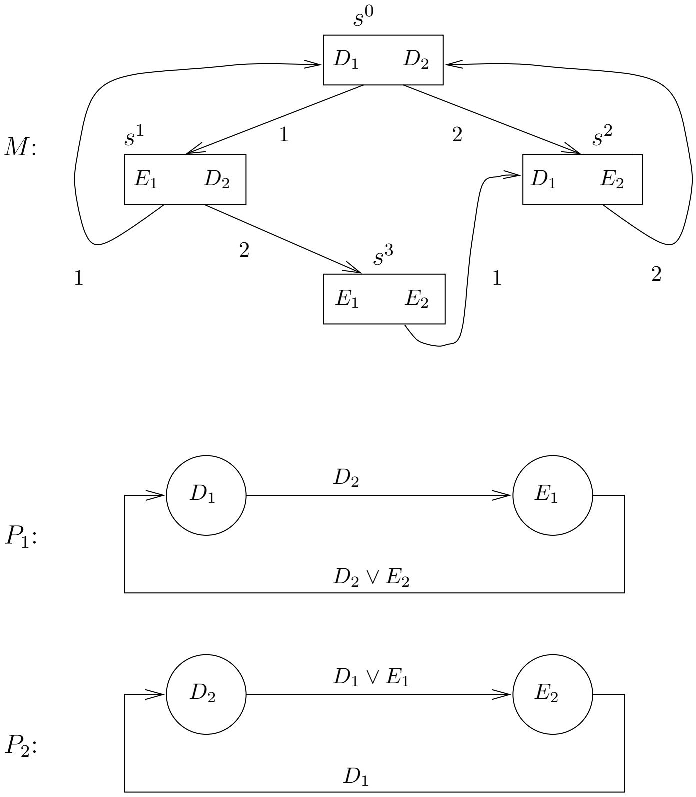


Fig. 1. Example program $P = P_1 \| P_2$ and its global-state transition diagram M.

Formally, we define the semantics of CTL formulae with respect to a Kripke structure $M = (S_0, S, R)$ consisting of

S, a countable set of global states.

$S_0 \subseteq S$, a countable set of initial states.

$R \subseteq S \times [1 : K] \times \mathcal{A} \times S$, a transition relation. R is partitioned into relations $R_1, \ldots, R_K$, where R_i gives the transitions of process i.

A *fullpath* is a maximal computation path, i.e., a path that is either infinite or ends in a state with no outgoing transitions. If π is a fullpath, then define $|\pi|$, the length of π, to be ω when π is infinite and k when π is finite and of the form $(s^0, \ldots, s^k)$. We use the usual notation for 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:

$$
\begin{array}{lll}
M, s^0 \models Q_i & \text{iff} & Q_i \in \mathcal{L}[s^0{\uparrow}i] \\
M, s^0 \models \neg f & \text{iff} & \text{not}(M, s^0 \models f) \\
M, s^0 \models f \wedge g & \text{iff} & M, s^0 \models f \text{ and } M, s^0 \models g \\
M, s^0 \models \mathsf{EX}_j f & \text{iff} & \text{for some state } t,\ (s^0, t) \in R_j \text{ and } M, t \models f
\end{array}
$$

$$
\begin{array}{lll}
M, s^0 \models \mathsf{A}[f\mathsf{U}g] & \text{iff} & \text{for all fullpaths } \pi = (s^0, s^1, \ldots) \text{ in } M \text{ that start in } s^0, \\
& & \quad \text{there exists } i \in [0 : |\pi|] \text{ such that} \\
& & \qquad M, s^i \models g \text{ and for all } j \in [1 : (i-1)]\text{:} \quad M, s^j \models f \\
M, s^0 \models \mathsf{E}[f\mathsf{U}g] & \text{iff} & \text{for some fullpath } \pi = (s^0, s^1, \ldots) \text{ in } M \text{ that starts in } s^0, \\
& & \quad \text{there exists } i \in [0 : |\pi|] \text{ such that} \\
& & \qquad M, s^i \models g \text{ and for all } j \in [1 : (i-1)]\text{:} \quad M, s^j \models f
\end{array}
$$

We say that a formula f is *satisfiable* if and only if there exists a structure M and state s of M such that $M, s \models f$. In this case, we say that M is a *model* of f. We say that a formula f is *valid* if and only if $M, s \models f$ for all structures M and states s of M.

We use the notation $M, U \models f$ as an abbreviation of $\forall s \in U : M, s \models f$, where $U \subseteq S$. We introduce the abbreviations $f \vee g$ for $\neg(\neg f \wedge \neg g)$, $f \Rightarrow g$ for $\neg f \vee g$, $f \equiv g$ for $(f \Rightarrow g) \wedge (g \Rightarrow f)$, $\mathsf{AF}f$ for $\mathsf{A}[\mathit{true}\mathsf{U}f]$, $\mathsf{EF}f$ for $\mathsf{E}[\mathit{true}\mathsf{U}f]$, $\mathsf{AG}f$ for $\neg\mathsf{EF}\neg f$, $\mathsf{EG}f$ for $\neg\mathsf{AF}\neg f$, $\mathsf{AY}_i f$ for $\neg\mathsf{EX}_i\neg f$, $\mathsf{EX}f$ for $\mathsf{EX}_1 f \vee \cdots \vee \mathsf{EX}_k f$, and $\mathsf{AY}f$ for $\mathsf{AY}_1 f \wedge \cdots \wedge \mathsf{AY}_k f$.

A formula of the form $\mathsf{A}[f\mathsf{U}g]$ or $\mathsf{E}[f\mathsf{U}g]$ 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 $\mathsf{A}[f\mathsf{U}g]$ ($\mathsf{E}[f\mathsf{U}g]$) 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 $\mathsf{AF}g$ and $\mathsf{EF}g$ are special cases of $\mathsf{A}[f\mathsf{U}g]$ and $\mathsf{E}[f\mathsf{U}g]$, respectively, they are also eventualities. In contrast, $\mathsf{AG}f$ and $\mathsf{EG}f$ are *invariance* formulae. An invariance corresponds to a safety property, since it asserts that whatever happens to occur (if anything) will meet certain conditions.

Following Definition 2.2.2, we annotate transitions in a Kripke structure with the index i of the process executing the transition, and the assignment statement A (if any) that P_i executes, e.g., $s \xrightarrow{i,A} t$. In the sequel, we assume, for all Kripke structures $M = (S_0, S, R)$, and all $s \xrightarrow{i,A} t \in R$: $\wedge j \in [1 : K] - \{i\} : s{\uparrow}j = t{\uparrow}j$ and $\langle s{\uparrow}\mathcal{SH}\rangle\, A\, \langle t{\uparrow}\mathcal{SH}\rangle$. This merely excludes Kripke structures that do not respect our model of concurrent computation. All Kripke structures generated by our synthesis method (and the method of Emerson and Clarke [1982]) satisfy this assumption.

2.4 The Specification Language

For the purposes of this paper, we restrict specifications to the sublogic of CTL whose formulae are finite conjunctions of the following terms:

—h, where $h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, specifies the initial states.[6]

—$\mathsf{AG}h$, where $h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, specifies *safety* properties, since $\mathsf{AG}h$ is satisfied by all initial states iff h is satisfied by all reachable states, i.e., h is invariant.

—$\mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$, where $p, q, r \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, specifies a temporal leads-to property, combined with an "unless" property: from any global state satisfying p, eventually a state satisfying r must be reached (along all outgoing fullpaths), and all intervening states must satisfy q.

[6] We denote various sets of propositional formulae by $\mathcal{LO}$ followed by a list of the atomic propositions and boolean operators that can be used in constructing the formulae.

—$\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, where $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$, specifies a *local structure* property; from every reachable state satisfying p_i, there exists an outgoing P_i-transition to a state satisfying q_i. In the synthesized program, such a transition (if it involves both a test and a set) will be broken down into a test transition followed by some set transitions. Hence, the synthesized program will not, in general, satisfy $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$; it will, however, satisfy $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i(p_i \vee q_i))$. For technical reasons, we require $p_i \not\equiv q_i$.

—$\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$, where $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$, specifies a *local structure* property; from every reachable state satisfying p_i, every outgoing P_i-transition is to a state satisfying q_i. For the same reason as given for $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, the synthesized program will not, in general, satisfy $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$; it will, however, satisfy $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$.

In addition, we assume that $\mathsf{AGEX}\mathit{true}$ is always a conjunct of the specification. That is, we restrict our attention to nonterminating concurrent programs.

An example specification is given below for the two-process mutual exclusion problem. In this specification, process P_i, $i \in \{1, 2\}$, is always in exactly one of three states N_i, T_i, C_i, which represent, respectively, the noncritical, trying, and critical code regions. A process in its noncritical state does not require the use of critical (shared) resources, and performs computations that can proceed in parallel with computations by the other processes. A process requiring a critical resource moves into the trying state. From there, it enters the critical state as soon as possible, provided that all constraints on the use of the critical resource are met. In the critical state, P_i has access to the critical resource. This specification is given by the following set of CTL formulae (which are implicitly conjoined):

(1) Initial State (both processes are initially in their noncritical region):
$N_1 \wedge N_2$

(2) It is always the case that any move P_1 makes from its noncritical region is into its trying region and such a move is always possible. Likewise for P_2:
$\mathsf{AG}(N_1 \Rightarrow (\mathsf{AY}_1 T_1 \wedge \mathsf{EX}_1 T_1))$
$\mathsf{AG}(N_2 \Rightarrow (\mathsf{AY}_2 T_2 \wedge \mathsf{EX}_2 T_2))$

(3) It is always the case that any move P_1 makes from its trying region is into its critical region. Likewise for P_2:
$\mathsf{AG}(T_1 \Rightarrow (\mathsf{AY}_1 C_1))$
$\mathsf{AG}(T_2 \Rightarrow (\mathsf{AY}_2 C_2))$

(4) It is always the case that any move P_1 makes from its critical region is into its noncritical region and such a move is always possible. Likewise for P_2:
$\mathsf{AG}(C_1 \Rightarrow (\mathsf{AY}_1 N_1 \wedge \mathsf{EX}_1 N_1))$
$\mathsf{AG}(C_2 \Rightarrow (\mathsf{AY}_2 N_2 \wedge \mathsf{EX}_2 N_2))$

(5) P_1 is always in exactly one of the states N_1, T_1, or C_1. Likewise for P_2:
$\mathsf{AG}(N_1 \equiv \neg(T_1 \vee C_1)) \wedge \mathsf{AG}(T_1 \equiv \neg(N_1 \vee C_1)) \wedge \mathsf{AG}(C_1 \equiv \neg(N_1 \vee T_1))$
$\mathsf{AG}(N_2 \equiv \neg(T_2 \vee C_2)) \wedge \mathsf{AG}(T_2 \equiv \neg(N_2 \vee C_2)) \wedge \mathsf{AG}(C_2 \equiv \neg(N_2 \vee T_2))$

(6) Every request for critical section entry by P_1, P_2 is eventually granted, i.e., P_1, P_2 do not starve:
$\mathsf{AG}(T_1 \Rightarrow \mathsf{AF}C_1)$
$\mathsf{AG}(T_2 \Rightarrow \mathsf{AF}C_2)$

(7) A transition by one process cannot cause a transition by another (interleaving model of concurrency):

$$\mathsf{AG}((N_1 \Rightarrow \mathsf{AY}_2 N_1) \land (N_2 \Rightarrow \mathsf{AY}_1 N_2))$$
$$\mathsf{AG}((T_1 \Rightarrow \mathsf{AY}_2 T_1) \land (T_2 \Rightarrow \mathsf{AY}_1 T_2))$$
$$\mathsf{AG}((C_1 \Rightarrow \mathsf{AY}_2 C_1) \land (C_2 \Rightarrow \mathsf{AY}_1 C_2))$$

(8) P_1, P_2 do not access critical resources together:

$$\mathsf{AG}(\neg(C_1 \land C_2))$$

(9) It is always the case that some process can move:

$$\mathsf{AGEX}\mathit{true}$$

Note that since $\mathsf{AF}f \equiv \mathsf{A}[\mathit{true}\mathsf{U}f]$, the above is within our specification language.

2.5 Correctness Properties of Programs

Let $M = (S_0, S, R)$ be the global-state transition diagram of program P. From the definitions of global-state transition diagram and Kripke structure given above, we see that M can be considered to be a Kripke structure. Hence, we consider a program P to be correct with respect to a specification f (expressed as a CTL formula) iff $M, S_0 \models f$. In other words, the specification must be true in all the initial states of the global-state transition diagram of the program. In this case, we say that program P *satisfies* specification f.

2.6 Synthesis of Programs

Our method builds on and extends the synthesis method of Emerson and Clarke [1982].[7] The method of Emerson and Clarke [1982] starts with a CTL specification f and applies a tableau-based decision procedure to f. If f is satisfiable, the decision procedure constructs a tableau from which a model M of f can be constructed. For example, Figure 2 shows the model generated for the two-process mutual exclusion specification given in Section 2.4. The model M can be regarded as the global-state transition diagram of a correct program, and this program can be extracted from M by "projecting" onto the individual processes. This extraction operation was described informally in Emerson and Clarke [1982]. To establish our technical results, we need a formal definition of program extraction. We proceed as follows.

Let s be a state. We use $s{\downarrow}i$ for $s - \{s_i\}$, i.e., s with its P_i-component removed. We define,

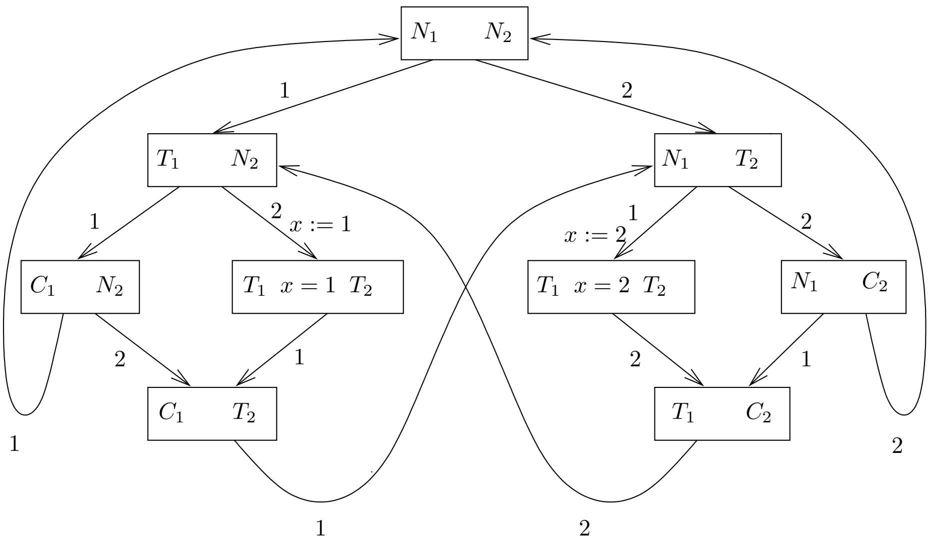
Definition 2.1 (*State-to-Formula Definition*).

$$\{\!\{s\}\!\} = \text{“}(\bigwedge_{s(Q)=\mathit{true}} Q) \ \land \ (\bigwedge_{s(Q)=\mathit{false}} \neg Q) \ \land \ (\bigwedge_{x \in \mathcal{SH}} x = s(x))\text{”}$$

where Q ranges over $\mathcal{AP}$

$$\{\!\{s{\downarrow}i\}\!\} = \text{“}(\bigwedge_{s(Q)=\mathit{true}} Q) \ \land \ (\bigwedge_{s(Q)=\mathit{false}} \neg Q) \ \land \ (\bigwedge_{x \in \mathcal{SH}} x = s(x))\text{”}$$

[7] While this paper is self-contained, the reader is referred to Emerson and Clarke [1982] for more detail.



The initial state set is { [N_1 N_2] }

Fig. 2. Global-state transition diagram for the two-process mutual exclusion problem produced by the synthesis method of Emerson and Clarke [1982].

where Q ranges over $\mathcal{AP} - \mathcal{AP}_i$

$$\{\!\!\{s_i\}\!\!\} = \text{“}(\bigwedge_{s(Q)=true} Q) \;\wedge\; (\bigwedge_{s(Q)=false} \neg Q)\text{”}$$

where Q ranges over $\mathcal{AP}_i$.

$\{\!\!\{s\}\!\!\}$ characterizes s in that $s \models \{\!\!\{s\}\!\!\}$, and $s' \not\models \{\!\!\{s\}\!\!\}$ for all states s' such that $s' \neq s$, i.e., it converts a state into a propositional formula. For example, $\{\!\!\{s^0\}\!\!\} = A_1 \wedge \neg B_1 \wedge A_2 \wedge \neg B_2$, where s^0 is the global state depicted in Figure 1.

From the global-state transition diagram definition (2.2.2), we see that a particular P_i-arc $(s_i, B \to A, t_i)$ generates a set of P_i-transitions; it generates one P_i-transition in every state s such that $s{\uparrow}i = s_i$ and $s(B) = \mathit{true}$. We group all the P_i-transitions generated by a single P_i-arc into a P_i*-family*:[8]

Definition 2.2 (P_i*-Family*). A P_i-family $\mathcal{F}$ in a Kripke structure $M = (S_0, S, R)$ is a maximal subset of R such that

(1) all members of $\mathcal{F}$ are P_i-transitions, and have the same label $\xrightarrow{i,A}$, and

(2) for any pair $s \xrightarrow{i,A} t, s' \xrightarrow{i,A} t'$ of members of $\mathcal{F}$: $s{\uparrow}i = s'{\uparrow}i$ and $t{\uparrow}i = t'{\uparrow}i$.

[8] We assume that every process P_i contains, between any pair of local states, at most one arc labeled with a given assignment statement. If process P_i contains two such arcs, say $(s_i, B1 \to A, t_i)$ and $(s_i, B2 \to A, t_i)$, then these can be combined into the single arc $(s_i, (B1 \vee B2) \to A, t_i)$.

If $s \xrightarrow{i,A} t \in \mathcal{F}$, then let $\mathcal{F}.start$, $\mathcal{F}.finish$, $\mathcal{F}.assig$, $\mathcal{F}.label$ denote $s{\uparrow}i$, $t{\uparrow}i$, A, and $\xrightarrow{i,A}$ respectively. Given that $T.begin$ denotes the source state of transition T, i.e., $T.begin = s$ for transition $T = s \xrightarrow{i,A} t$, let $\mathcal{F}.guard$ denote $\bigvee_{T \in \mathcal{F}} \{\!\{(T.begin){\downarrow}i\}\!\}$.

When P_i is understood from context, or when we wish to refer to a P_i-family for an unspecified P_i, we will use the term "family," i.e., we drop the prefix "P_i-". $\mathcal{F}.guard$ is equivalent to the guard B of the P_i-arc $(s{\uparrow}i, B \to A, t{\uparrow}i)$ which generates $\mathcal{F}$.[9] The P_i-family definition allows us to give a succinct definition for the operation of extracting a program from a Kripke structure. This definition formalizes the extraction technique of Emerson and Clarke [1982].

Definition 2.3 (*Program Extraction Definition*). Let $M = (S_0, S, R)$ be an arbitrary Kripke structure. Then the program $P = P_1 \| \cdots \| P_K$ extracted from M is given by

$(s_i, B \to A, t_i) \in P_i$ iff
there exists a P_i-family $\mathcal{F}$ in M such that
$\mathcal{F}.start = s_i$, $\mathcal{F}.finish = t_i$, $\mathcal{F}.assig = A$, $\mathcal{F}.guard = B$.

For example, the Kripke structure shown in Figure 2 contains the P_1-families

$$
\begin{aligned}
&\{\ [N_1\, N_2] \xrightarrow{1} [T_1\, N_2],\ [N_1\, C_2] \xrightarrow{1} [T_1\, C_2]\ \},\\
&\{\ [N_1\, T_2] \xrightarrow{1, x:=2} [T_1\, x = 2\, T_2]\ \},\\
&\{\ [T_1\, N_2] \xrightarrow{1} [C_1\, N_2],\ [T_1\, x = 1\, T_2] \xrightarrow{1} [C_1\, T_2],\ \},\\
&\{\ [C_1\, N_2] \xrightarrow{1} [N_1\, N_2],\ [C_1\, T_2] \xrightarrow{1} [N_1\, T_2]\ \},
\end{aligned}
$$

and the P_2 families

$$
\begin{aligned}
&\{\ [N_1\, N_2] \xrightarrow{2} [N_1\, T_2],\ [C_1\, N_2] \xrightarrow{2} [C_1\, T_2]\ \},\\
&\{\ [T_1\, N_2] \xrightarrow{2, x:=1} [T_1\, x = 1\, T_2]\ \},\\
&\{\ [N_1\, T_2] \xrightarrow{2} [N_1\, C_2],\ [T_1\, x = 2\, T_2] \xrightarrow{2} [T_1\, C_2],\ \},\\
&\{\ [N_1\, C_2] \xrightarrow{2} [N_1\, N_2],\ [T_1\, C_2] \xrightarrow{2} [T_1\, N_2]\ \}.
\end{aligned}
$$

Applying Definition 2.3, the above families give rise to the following P_1-arcs, respectively

$$
\begin{aligned}
&(N_1, N_2 \vee C_2 \to skip, T_1),\\
&(N_1, T_2 \to x := 2, T_1),\\
&(T_1, N_2 \vee (T_2 \wedge x = 1) \to skip, C_1),\\
&(C_1, N_2 \vee T_2 \to skip, N_1),
\end{aligned}
$$

and the following P_2-arcs, respectively,

$$
\begin{aligned}
&(N_2, N_1 \vee C_1 \to skip, T_2),\\
&(N_2, T_1 \to x := 1, T_2),
\end{aligned}
$$

[9] Equivalence is with respect to the Kripke structure M. In other words, $\mathcal{F}.guard$ and B agree on every reachable global state u in M whose projection onto P_i is $\mathcal{F}.start$, the local start state for a transition in $\mathcal{F}$. That is, $M, S^r \models \{\!\{\mathcal{F}.start\}\!\} \Rightarrow (B \equiv \mathcal{F}.guard)$, where S^r is the set of reachable states of M. Another way of stating this is $\{\!\{\mathcal{F}.start\}\!\}^M \cap \mathcal{F}.guard^M = \{\!\{\mathcal{F}.start\}\!\}^M \cap B^M$, where $f^M = \{s \mid s \models f$ and s is a reachable state of $M\}$.

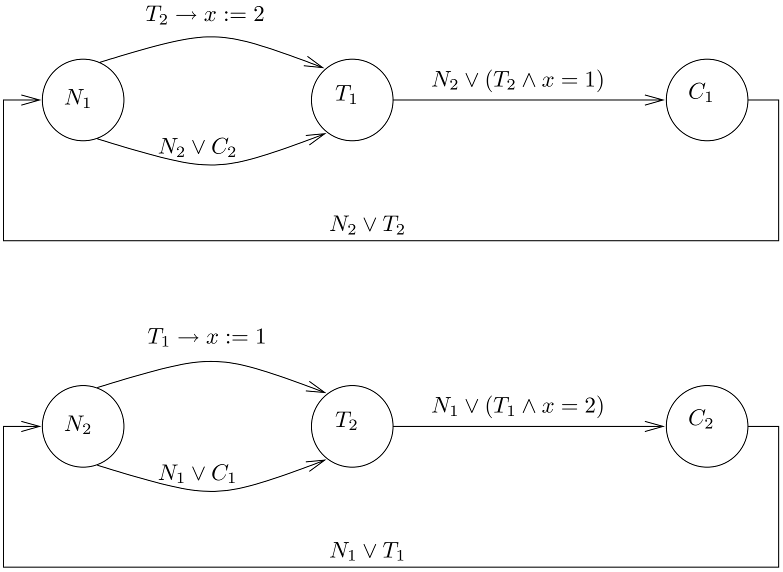


Fig. 3. Program for the two-process mutual exclusion problem extracted from the Kripke structure of Figure 2.

$$(T_2, N_1 \vee (T_1 \wedge x = 2) \rightarrow skip, C_2),$$
$$(C_2, N_1 \vee T_1 \rightarrow skip, N_2).$$

The resulting program is depicted in Figure 3. We remark that this program contains large-grain atomicity operations. For example, one of the arcs from N_1 to T_1 is labeled with the guarded command $T_2 \rightarrow x := 2$. Thus the atomic proposition T_2 must be tested, and then the assignment $x := 2$ must be performed. In addition, the values assigned to the atomic propositions N_1, T_1 must be changed from *true*, *false* respectively to *false*, *true* respectively. So, overall, a test followed by a multiple assignment must be performed as a single atomic operation.

Definition 2.3 guarantees that the execution of program P generates the same structure M that P was extracted from, i.e., the program extraction and global-state transition diagram generation operations are "inverses." In particular, the use of $\mathcal{F}.guard = \bigvee_{T \in \mathcal{F}} \{\!\!\{(T.begin)\!\downarrow\! i\}\!\!\}$ as the guard B of the arc $(s_i, B \rightarrow A, t_i)$ corresponding to family $\mathcal{F}$ ensures that all and only the transitions in $\mathcal{F}$ are generated by the execution of $(s_i, B \rightarrow A, t_i)$. When P_i is in state s_i, the disjunct $\{\!\!\{(T.begin)\!\downarrow\! i\}\!\!\}$ of $\mathcal{F}.guard$ evaluates to *true* if and only if the current global state is $T.begin$. Hence, $\{\!\!\{(T.begin)\!\downarrow\! i\}\!\!\}$ contributes the transition T (and only the transition T) to the global-state transition diagram of P. Hence, $\mathcal{F}.guard$ contributes exactly the transitions in $\mathcal{F}$, as desired.

We note that the model in Figure 2 contains a shared variable x. This variable serves to disambiguate the states $[T_1\ x = 1\ T_2]$ and $[T_1\ x = 2\ T_2]$, which are propositionally equivalent, but differ temporally, since $[T_1\ x = 1\ T_2]$ satisfies $\mathsf{EX}_1 true$, while $[T_1\ x = 2\ T_2]$ satisfies $\mathsf{EX}_2 true$. Without x, the extracted program would generate a global-state transition diagram in which these states are "merged" into a single global state $[T_1\ T_2]$. This is because the extracted program depends only on the propositional component of the states, and does not take the temporal information that labels the states into account (see Emerson and Clarke [1982]). In this global-state transition diagram, there is a cycle $[N_1\ T_2] \xrightarrow{1} [T_1\ T_2] \xrightarrow{1} [C_1\ T_2] \xrightarrow{1} [N_1\ T_2]$. This cycle causes violation of the absence-of-starvation specification $\mathsf{AG}(T_2 \Rightarrow \mathsf{AF}C_2)$. Hence we see that it is necessary that propositionally identical states be disambiguated in some way, e.g., by the use of shared variables.

In summary, we see that there is a correspondence between P_i-arcs in a program P and P_i-families in the global-state transition diagram M of P. A P_i-arc AR_i in P gives rise to a P_i-family of transitions in M which represent all the possible executions of AR_i, i.e., the executions of AR_i in all the global states that satisfy the guard of AR_i, and where local control of P_i is "at" the start state of AR_i, as specified by the global-state transition diagram definition (2.2.2). Also, a P_i-family $\mathcal{F}$ of transitions in M gives rise to a single P_i-arc when P is extracted from M, the guard of this arc being the disjunction of all global states s (with the P_i-component removed, i.e., $s{\downarrow}i$, and then converted to a formula, i.e., $\{\!|s{\downarrow}i|\!\}$) which are source states of some transition in $\mathcal{F}$.

3. ATOMIC READ/WRITE SYNTAX AND SEMANTICS

3.1 Syntactic Characterization of Atomic Read/Write Programs

We recall that a P_i-transition of a program $P = P_1 \| \cdots \| P_K$ may, in general, change one or more atomic propositions in $\mathcal{AP}_i$ and/or shared variables. In effect, the P_i-transition executes a multiple assignment. Since we aim to synthesize programs for an atomic read/write model of computation, such a P_i-transition must be decomposed into a series of P_i-transitions, each of which executes a single assignment.

Intuitively, $P = P_1 \| \cdots \| P_K$ is an atomic read/write program iff the execution of every arc (of every process of) P requires only a single atomic read operation or a single atomic write operation. We formalize this by defining a set of *syntactic arc attributes* (Definition 3.1.2 below). These attributes characterize an arc in terms of the atomic read/write operations that are required to implement it. Some of the clauses of Definition 3.1.2 have an accompanying (indented) paragraph of explanatory text.

Definition 3.1.1 (*Syntactic Equality*). We use $\doteq$ to denote syntactic equality. i.e., $B \doteq \mathit{true}$ means that B is the constant *true*. If B is $N_1 \vee \neg N_1$, for example, then $B \not\doteq \mathit{true}$.

Definition 3.1.2 (*Syntactic Arc Attributes*). We define the following seven attributes of synchronization skeleton arcs.

—An arc $(s_i, B \rightarrow A, t_i)$ is *guarded* iff $B \not\doteq \mathit{true}$. A guarded arc requires one or more read operations for its implementation, since the guard, in general, refers to shared variables and/or atomic propositions (of other processes), which must be read in order that the guard be evaluated.

—An arc $(s_i, B \rightarrow A, t_i)$ is *unguarded* iff $B \doteq \mathit{true}$. An unguarded arc requires no read operations for its implementation, since it can be executed unconditionally when local control is "at" its start state.

—An arc $(s_i, B \rightarrow A, t_i)$ is *single-writing* iff $A \doteq skip$ and $\mathcal{L}[s_i] \neq \mathcal{L}[t_i]$, or $A \doteq$ "$x := c$" (for some shared variable x and $c \in D_x$) and $\mathcal{L}[s_i] = \mathcal{L}[t_i]$. In the first case, a single write operation is needed to set the values of all atomic propositions in $\mathcal{AP}_i$ to those assigned by t_i. Since these atomic propositions are combined into the externally visible location counter L_i, this requires only a single write operation. In the second case, a single write operation sets the shared variable x to the constant c.

—An arc $(s_i, B \rightarrow //_{m \in [1:n]}\ x^m := c^m, t_i)$ is *multiple-writing* iff $n > 1$, or ($n > 0$ and $\mathcal{L}[s_i] \neq \mathcal{L}[t_i]$). In the first case, one write operation is needed to set the values of all atomic propositions in $\mathcal{AP}_i$ to those assigned by t_i, and at least one write operation is needed to perform $//_{m \in [1:n]}\ x^m := c^m$, since we have $n > 0$. In the second case, $n > 1$ write operations are needed to perform $//_{m \in [1:n]}\ x^m := c^m$. Note that an arc $(s_i, B \rightarrow skip, t_i)$ is never multiple-writing.

—An arc is *writing* iff it is either single-writing or multiple-writing.

—An arc $(s_i, B \rightarrow skip, t_i)$ is *nonwriting* iff $\mathcal{L}[s_i] = \mathcal{L}[t_i]$. An arc is nonwriting if and only if it is not writing.

—An arc is *test-and-set* iff it is both guarded and writing.

Define a *simple term* to be a formula of the form $Q_i \in L_i$, (where $Q_i \in \mathcal{AP}_i, i \in [1:K]$), or of the form $x = c$, (where $x \in \mathcal{SH}, c \in D_x$). In an atomic read/write model, a simple term can be used as the guard of an arc, since checking that a simple term evaluates to *true* (and therefore that the arc can be executed) can be done using a single atomic read operation (which reads the single shared variable or externally visible location counter that the simple term refers to, depending on its form). Likewise, a disjunction of simple terms can be used as the guard of an arc, since checking that a disjunction evaluates to *true* reduces to checking that one of its disjuncts evaluates to *true*. However, a conjunction of terms (simple or otherwise) cannot be used as the guard of an arc in a straightforward manner, since checking that a conjunction evaluates to *true* requires us to check that all conjuncts evaluate to *true* simultaneously. This requires the simultaneous reading of all the shared variables and location counters that are mentioned in any conjunct, and the atomic read/write model does not permit such "multiple" reads.

Definition 3.1.3 (*Single-Reading*). An arc $(s_i, B \rightarrow A, t_i)$ is *single-reading* iff B is a disjunction of simple terms.

Note, that in the synchronization skeleton model, P_i does not need to read its own local state s_i, since the model requires that local control of P_i is "at" s_i before $(s_i, B \rightarrow A, t_i)$ can be executed. (In Section 6, we show how to translate our synchronization skeleton programs into a model based on atomic read/write registers. In this model, the local state s_i does have to be read by process P_i.)

Definition 3.1.4 (*Atomic Read/Write Program*). P is an atomic read/write program iff P is a program such that every arc in P is either single-reading and nonwriting, or unguarded and single-writing.

3.2 Numbered Local States

By Definition 3.1.4 above, $P = P_1 \| \cdots \| P_K$ is in atomic read/write form iff every arc of every process $P_1, \ldots, P_K$ either reads a single variable or writes a single variable.

In our framework, there are two types of variables: (1) shared variables, which every process can read/write, and (2) the externally visible location counter, which the owning process can write and which every other process can read. Consider an arc of P_i that reads a single variable. This arc must have the form $(s_i, B \rightarrow \mathit{skip}, t_i)$, where B references the variable. Also, $\mathcal{L}[s_i] = \mathcal{L}[t_i]$, since no atomic propositions in $\mathcal{AP}_i$ can be modified. However, it is clear that, in general, s_i and t_i do not denote the same i-state (i.e., $s_i \neq t_i$), since P_i makes the transition from s_i to t_i based on B evaluating to *true*. If s_i and t_i were the same i-state, then executing the arc $(s_i, B \rightarrow \mathit{skip}, t_i)$ would not change anything, i.e., there would be no point in testing the truth of B, since the final local state is the same regardless of the outcome of the test. Thus, the value read has no effect on subsequent execution.[10] We conclude that in general s_i and t_i are different i-states that happen to assign the same truth values to all atomic propositions in $\mathcal{AP}_i$, and therefore have the same externally visible location counter. Now, when the current i-state is t_i instead of s_i, this records the fact that B evaluated to *true*. Hence, the read operation impacts the subsequent execution.

Since the externally visible location counter does not distinguish s_i and t_i, we introduce a function *num*, from the set of i-states to the natural numbers, that assigns a natural number to each i-state. The idea is that i-states with the same externally visible location counter (i.e., local atomic proposition valuations) are assigned different numbers:

$$\mathcal{L}[s_i] = \mathcal{L}[t_i] \land s_i \neq t_i \Rightarrow \mathit{num}(s_i) \neq \mathit{num}(t_i). \qquad \text{(NUM)}$$

Since another way of writing (NUM) is

$$\mathcal{L}[s_i] = \mathcal{L}[t_i] \land \mathit{num}(s_i) = \mathit{num}(t_i) \Rightarrow s_i = t_i,$$

we can take the pair $(\mathcal{L}[s_i], \mathit{num}(s_i))$ as uniquely defining s_i, i.e., as giving the location counter (which, in effect, points to the "current" local state). $\mathcal{L}[s_i]$ is the externally visible component of the location counter (readable by other processes), and $\mathit{num}(s_i)$ is the internal component of the location counter (not readable by other processes). In examples, we shall not give local states explicit names (e.g., s_i, t_i, etc.) but will use the combination of $\mathcal{L}[s_i]$ and $\mathit{num}(s_i)$ to identify local states. We shall usually indicate the number assigned to an i-state as a superscript, e.g., s_i^1, s_i^2, etc. Since a global state is a tuple of local states and shared variable values, we shall not name global states in examples. Global states are uniquely identified by their constituent local states and shared variable values.

3.3 Atomic Read/Write Kripke Structures

In Sections 2.2 and 2.6, we dealt with the relationship between programs and Kripke structures. Since an arbitrary Kripke structure can contain transitions that assign to several shared variables and test the global state, it is clear that the programs extracted from such a structure must contain "test-and-set" operations of a large grain of atomicity. Since the set of all (finite state) programs corresponds (via

[10] Note that we are not claiming that programs in our model do not use "busy waiting." Busy waiting, however, comes into the picture when we implement the synchronization skeleton model. In an implementation, busy waiting would usually be needed in order to repeatedly test the guard of an arc, so that the arc can be executed when the guard evaluates to *true*.

extraction in one direction and global-state transition diagram generation in the other) to the set of all (finite state) Kripke structures, it follows from the above that the set of atomic read/write programs corresponds to a proper subset of the set of (finite state) Kripke structures. We now define this subset: the atomic read/write Kripke structures. We also define the operation for extracting an atomic read/write program from an atomic read/write Kripke structure.

Since there is a correspondence between P_i-arcs and P_i-families, the syntactic arc attributes can be extended to families merely by applying Definition 3.1.2 to the arc corresponding to the family. We do this for the "writing" attributes:

Definition 3.3.1 (*Family Write Attributes*). A family $\mathcal{F}$ is writing, nonwriting single-writing, multiple-writing, iff the arc $(\mathcal{F}.start, \mathcal{F}.guard \rightarrow \mathcal{F}.assig, \mathcal{F}.finish)$ is writing, nonwriting, single-writing, multiple-writing, respectively.

For the "guarding" attributes, we find it technically convenient to use the following definition instead.

Definition 3.3.2 (*Unguarded Family, Guarded Family*). A P_i-family $\mathcal{F}$ is *unguarded* in Kripke structure $M = (S_0, S, R)$ iff:

$$\begin{array}{l}\text{for all reachable states } s \text{ in } M \text{ such that } s{\uparrow}i = \mathcal{F}.start,\\ \qquad\text{there exists a transition } T \in \mathcal{F} \text{ such that } T.begin = s\end{array}$$

A P_i-family $\mathcal{F}$ is *guarded in* M iff $\mathcal{F}$ is not unguarded in M.

In other words, an unguarded family contains a transition starting in every reachable state s whose P_i-projection $s{\uparrow}i$ is equal to the start state of the family. Hence the P_i-arc $(\mathcal{F}.start, \mathcal{F}.guard \rightarrow \mathcal{F}.assig, \mathcal{F}.finish)$ extracted from $\mathcal{F}$ can be executed in every state s such that $s{\uparrow}i = \mathcal{F}.start$. Hence $s \models \{\!\!\{\mathcal{F}.start\}\!\!\}$ implies $s \models \mathcal{F}.guard$ for every reachable state s. We can rewrite this as $s \models \{\!\!\{\mathcal{F}.start\}\!\!\} \Rightarrow (\mathcal{F}.guard \equiv \mathit{true})$. Thus $\mathcal{F}.guard$ can be replaced by *true*, and so the arc extracted from an unguarded family is also unguarded.[11] Also, if AR_i is an unguarded arc in program P, then the P_i-family that AR_i generates in the global-state transition diagram of P (according to the global-state transition diagram definition (2.2.2)), is also unguarded.

Consider a guarded P_i-family $\mathcal{F}$ in M. If we apply the program extraction definition (2.3) to extract a program from M, then the P_i-arc corresponding to $\mathcal{F}$ will have the guard $\mathcal{F}.guard$, which, by the P_i-family definition (2.2), is $\bigvee_{T \in \mathcal{F}} \{\!\!\{T.begin{\downarrow}i\}\!\!\}$. By the state-to-formula definition (2.1), $\{\!\!\{T.begin{\downarrow}i\}\!\!\}$ is a conjunction of simple terms. Thus we see that $\mathcal{F}.guard$ is in disjunctive normal form, with each disjunct being a conjunction of simple terms. The atomic read/write program definition (3.1.4) requires that every guard in an atomic read/write program be a disjunction of simple terms. Hence, to extract an atomic read/write program from M, we must replace $\mathcal{F}.guard$ by an equivalent[12] guard which is a disjunction of simple terms. If this can be done, then the P_i-family $\mathcal{F}$ can be implemented by a single-reading arc. We call such a family a *single-reading* family:

[11] In fact $\mathcal{F}.guard$ would be replaced by *true* in the *guard simplification* step of (Phase 1 of) the synthesis method. See Section 4.1.4.

[12] Equivalence is with respect to the Kripke structure M—see footnote 9.

Definition 3.3.3 (*Single-Reading Family*). A family $\mathcal{F}$ of a Kripke structure M is *single-reading* iff there exist subsets $\mathcal{F}_1, \ldots, \mathcal{F}_n$ of $\mathcal{F}$ such that $\mathcal{F} = \bigcup_{k \in [1:n]} \mathcal{F}_k$, and, for each $\mathcal{F}_k$, $(k \in [1:n])$, there exists a simple term b_k such that

$$\text{for all } T \in \mathcal{F}_k^r,\ T.begin(b_k) = true, \text{ and} \qquad \text{(G1)}$$

$$\begin{array}{l}\text{for all reachable states } s \text{ in } M \text{ such that } s{\uparrow}i = \mathcal{F}.start, \\ \quad \text{if } s \text{ is not the start state of some transition in } \mathcal{F}_k, \text{ then } s(b_k) = \mathit{false}, \end{array} \qquad \text{(G2)}$$

where $\mathcal{F}_k^r$ is the set of all reachable transitions in $\mathcal{F}_k$.

In other words, b_k is true in all states that are the begin state of some reachable transition in $\mathcal{F}_k$, and is false in any reachable state whose P_i-projection is the start state of $\mathcal{F}$, but which is not the begin state of some transition in $\mathcal{F}_k$. Thus, including b_k as a disjunct in the guard of the arc corresponding to $\mathcal{F}$ takes into account exactly the reachable transitions in $\mathcal{F}_k$. Hence, the guard $\bigvee_{k \in [1:n]} b_k$ takes into account all reachable transitions in $\mathcal{F}$ (since $\mathcal{F} = \bigcup_{k \in [1:n]} \mathcal{F}_k$). Furthermore, requiring that b_k be false in every state s such that $s{\uparrow}i = \mathcal{F}.start$ and such that s is not the start state of some transition in $\mathcal{F}_k$ ensures that no "extra" transitions are generated by the extracted programs (i.e., no transitions that are not present in the reachable portion of the Kripke structure from which the program is extracted). Thus $\bigvee_{k \in [1:n]} b_k$ is a suitable guard for the arc corresponding to $\mathcal{F}$. Also, $\bigvee_{k \in [1:n]} b_k$ is a disjunction of simple terms, as required.

We can now define atomic read/write Kripke structures. From such a structure, it is always possible to extract an atomic read/write program. Note how this definition parallels that of atomic read/write programs (Definition 3.1.4 above).

Definition 3.3.4 (*Atomic Read/Write Kripke Structure*). $M = (S_0, S, R)$ is an atomic read/write Kripke structure iff M is a Kripke structure such that every family in M is either single-reading and nonwriting, or unguarded and single-writing.

3.4 Extracting Atomic Read/Write Programs from Atomic Read/Write Kripke Structures

Given an atomic read/write Kripke structure M, an atomic read/write program P can be extracted from M according to the following definition, which is a modification of the program extraction definition (2.3).

Definition 3.4.1 (*Atomic Read/Write Program Extraction Definition*). Let M be an atomic read/write Kripke structure. Then, the program $P = P_1 \| \cdots \| P_K$ extracted from M is as follows:

$(s_i, B \to skip, t_i) \in P_i$ iff
there exists a single-reading and nonwriting P_i-family $\mathcal{F}$ in M such that
  $s_i = \mathcal{F}.start$, $t_i = \mathcal{F}.finish$, $B = \bigvee_{k \in [1:n]} b_k$, and there exist
  subsets $\mathcal{F}_1, \ldots, \mathcal{F}_n$ of $\mathcal{F}$ such that
    $\mathcal{F}_1, \ldots, \mathcal{F}_n, b_1, \ldots, b_n$ satisfy Definition 3.3.3

$(s_i, true \to A, t_i) \in P_i$ iff
there exists an unguarded, single-writing P_i-family $\mathcal{F}$ in M such that
  $s_i = \mathcal{F}.start$, $t_i = \mathcal{F}.finish$, and $A = \mathcal{F}.assig$

Proposition 3.4.2. *If P has been extracted from an atomic read/write Kripke structure M according to Definition 3.4.1, then P is an atomic read/write program.*

Proof. By Definition 3.4.1, the only arcs in P are of the form $(s_i, B \rightarrow \mathit{skip}, t_i)$ or of the form $(s_i, \mathit{true} \rightarrow A, t_i)$. For arcs of the form $(s_i, B \rightarrow \mathit{skip}, t_i)$, B is a disjunction of simple terms, by Definition 3.4.1 and Definition 3.3.3. Hence, such arcs are single-reading and nonwriting. For arcs AR_i of the form $(s_i, \mathit{true} \rightarrow A, t_i)$, the corresponding family $\mathcal{F}$ is single-writing, by Definition 3.4.1. Thus, by the family write attributes definition (3.3.1), AR_i is single-writing (since $s_i = \mathcal{F}.start$, $t_i = \mathcal{F}.finish$, and $A = \mathcal{F}.assig$). Hence AR_i is unguarded and single-writing. Thus, every arc in P is either single-reading and nonwriting, or unguarded and single-writing. Hence, by the atomic read/write program extraction definition (3.4.1), P is an atomic read/write program. □

The next lemma establishes that the atomic read/write extraction operation is "faithful" in that the global-state transition diagram of the extracted program is the same as the reachable portion of the original Kripke structure from which the program was extracted.

Lemma 3.4.3 (Correct Extraction). *Let $M^1 = (S_0, S, R^1)$ be an atomic read/write Kripke structure. Furthermore, let $P = P_1 \| \cdots \| P_K$ be an atomic read/write program extracted from M^1 using Definition 3.4.1, and let $M^2 = (S_0, S, R^2)$ be the global-state transition diagram of P obtained by applying Definition 2.2.2 to P. If a state s is reachable in both M^1 and M^2, then*

$$s \xrightarrow{i,A} t \in R^1 \text{ iff } s \xrightarrow{i,A} t \in R^2$$

for all i, A, t.

Proposition 3.4.4. *Let $M^1 = (S_0, S, R^1)$ be an atomic read/write Kripke structure. Furthermore, let P be an atomic read/write program extracted from M^1 using Definition 3.4.1, and let $M^2 = (S_0, S, R^2)$ be the global-state transition diagram of P obtained by applying Definition 2.2.2 to P. Let f be an arbitrary formula of* CTL. *Then*

$$M^1, S_0 \models f \text{ iff } M^2, S_0 \models f.$$

4. THE SYNTHESIS METHOD

Let f be a specification, expressed in CTL, for a concurrent program. Roughly speaking, we proceed as follows. We apply the CTL decision procedure of Emerson and Clarke [1982] to f. If f is satisfiable, then the decision procedure yields a model M of f. M can be viewed as the global-state transition diagram of a program P that satisfies f, and P can be extracted from M via the program extraction definition (2.3). In general, P will contain arbitrarily large grain test-and-set operations. We decompose these operations into single atomic read and single atomic write operations. The decomposition is straightforward and syntactic in nature; a test-and-set operation is decomposed into a test operation followed by a (multiple-)write operation, and a multiple-write operation is decomposed into a set of sequences of single-write operations which express all the possible serializations of the multiple-write operation. This decomposition may, in general, introduce new behaviors that violate the specification. We deal with this by generating the global-state transition diagram of the decomposed program, and then deleting all the portions of this diagram that are inconsistent with the specification. If some initial state is not deleted, then, from the resulting structure, an atomic read/write program that satisfies the specification can be extracted.

Our method will be presented as a sequence of phases, with each phase (except Phase 1) taking as input the output of the previous phase:

Phase 1 Extract a correct high-atomicity program from M, a finite-state model of the CTL problem specification.

Phase 2 Decompose the high-atomicity program into an atomic read/write program by syntactically decomposing each arc of each process.

Phase 3 Generate the global-state transition diagram of the atomic read/write program.

Phase 4 Delete portions of the global-state transition diagram that violate the specification.

Phase 5 From the resulting global-state transition diagram, extract an atomic read/write program that satisfies the specification.

In Phase 1, M could be given directly, as a "specification automaton," or could be produced from the CTL specification by the synthesis method of Emerson and Clarke [1982], whose application could then be considered to be "Phase 0." For technical reasons, we assume that M is in reachable form. Since only the reachable portion of a Kripke structure affects the behavior of an extracted program, this assumption does not restrict our synthesis method in any way. Furthermore, in practice we would wish to remove the unreachable portion of a Kripke structure simply as a matter of efficiency.[13] Note also that the externally visible location counter (which encodes the atomic propositions) is not introduced until Phase 2; Phase 1 deals directly with the atomic propositions.

4.1 Phase 1: Synthesize a Correct High-Atomicity Program

The objective of this phase is to synthesize an initial program that satisfies the specification f, but is not necessarily in atomic read/write form.

4.1.1 *Step 1.1: Derive the Initial Kripke Structure.* Our method accomodates a variety of problem specification techniques. If the problem specification is given as a CTL formula f, then we can apply the CTL decision procedure of Emerson and Clarke [1982] to f. If f is unsatisfiable, then no program can satisfy f. Otherwise, f is satisfiable, and the decision procedure yields a Kripke structure $M = (S_0, S, R)$ such that $M, S_0 \models f$, i.e., M is a model of f.

Alternatively, we can specify M directly, i.e., we view M as a state-machine whose executions constitute a set of "acceptable" executions. Examples of this technique are Lynch and Tuttle [1987], Lynch and Vaandrager [1995]. Even if M is given directly, our method still requires a CTL specification f which expresses the acceptable behaviors. This is because the decomposition performed in Phase 2

[13] If we wish to take *fault tolerance* or *self-stabilization* [Arora 1992; Arora et al. 1998] properties into account, then the unreachable portions do have to be considered. Since we do not consider such properties in this paper, we ignore "unreachable behavior."

Given a finite-state machine speficiation M, extract the corresponding CTL frame specification as the conjunction of all the formulae specified by the following rules.

Initial state: $\bigvee_{s \text{ is an initial state}} \{\!\{s\}\!\}$

Local AY-structure of each process: for all P_i: $\mathsf{AG}(\{\!\{s_i\}\!\} \Rightarrow \mathsf{AY}_i \bigvee_\ell \{\!\{t_i^\ell\}\!\})$ where t_i^ℓ ranges over all the P_i-states such that some arc of P_i goes from s_i to t_i^ℓ.

Local EX-structure of each process: for all P_i: $\mathsf{AG}(\{\!\{s_i\}\!\} \Rightarrow \mathsf{EX}_i \{\!\{t_i\}\!\})$ if, in M, every global state s with $s{\uparrow}i = s_i$ has some successor state t reached by a P_i-transition such that $t{\uparrow}i = t_i$.

The interleaving model: for all $P_i, P_j, i \neq j$ and $Q_i \in \mathcal{AP}_i$: $\mathsf{AG}(Q_i \Rightarrow \mathsf{AY}_j Q_i)$

Deadlock-freedom: $\mathsf{AGEX}\mathit{true}$

Fig. 4. Extracting the CTL frame specification from a finite-state machine specification.

may introduce new and undesirable behaviors, which have to be pruned out. The only way to determine which behaviors are undesirable is to have a specification against which to judge the new behaviors introduced by Phase 2. We have chosen to use a restricted subset of CTL for the purpose of writing these specifications. The main advantage is that the modifications required to eliminate the undesirable behaviors (and thereby restore the satisfaction of the specification) are reasonably straightforward (cf. the deletion rules in Section 4.4).

If M is given directly, then the accompanying CTL specification is usually significantly shorter than if the specification were given only as a CTL formula. This is because many of the conjuncts of a CTL specification are "frame specifications" which specify the initial state (e.g., clause 1 in the CTL specification of two-process mutual exclusion given in Section 2.4), the local structure of each process (clauses 2, 3, 4, and 5), the interleaving model (clause 7), and deadlock-freedom (clause 9). When M is given, such frame specifications can be extracted mechanically from M, as shown in Figure 4. They are then conjoined to the CTL specification accompanying M, and used as the specification for the purposes of the remaining steps in our synthesis method. (Except that the interleaving model clause is not needed, since it is implicitly satisfied by our operational semantics. The need for this clause arises in the Emerson and Clarke [1982] synthesis method.)

For example, looking at the CTL specification of two-process mutual exclusion given in Section 2.4, we see that clauses 8 and 6 express the crucial safety and liveness properties required of a solution to mutual exclusion. Thus, an alternative specification of mutual exclusion would be the Kripke structure of Figure 2, together with the (much shorter) CTL formula $\mathsf{AG}(\neg(C_1 \wedge C_2)) \wedge \mathsf{AG}(T_1 \Rightarrow \mathsf{AF}C_1)$ $\wedge$ $\mathsf{AG}(T_2 \Rightarrow \mathsf{AF}C_2)$.

We note, that when M and f are both given, then we do not require $M, s^0 \models f$ for every initial state s^0 of M. In other words, the given finite-state machine is not required to be "correct." Thus, our method provides both debugging and refinement, in this case.

4.1.2 *Step 1.2: Replicate Multiple Assignments.* First, we transform $M = (S_0, S, R)$ into an "equivalent" Kripke structure $M' = (S'_0, S, R')$ such that every multiple assignment $//_{m\in[1:n]}\ x^m := c^m$ in M is replicated along all "compatible" transitions. This has the desirable effect of weakening the guard of the assignment $//_{m\in[1:n]}\ x^m := c^m$ in the program extracted from M'.

Let $//_{m \in [1:n]}\ x^m := c^m$ be a multiple assignment that labels some P_i-transition $s \xrightarrow{i} t$ in M. We replicate this assignment along every P_i-transition $u \xrightarrow{i,A} v$ in M such that $u{\uparrow}i = s{\uparrow}i, v{\uparrow}i = t{\uparrow}i$, (i.e., every P_i-transition which takes P_i from the same (local) start state to the same (local) finish state) and A does not assign to any of $x^1, \ldots, x^n$. After this replication is performed (for all multiple assignments in M), it is possible that some states in M end up with two or more different values for the same shared variable, due to the extra assignments introduced. We therefore apply the following "propagation rules" (which "propagate" the resulting values of all shared variables) repeatedly to M, until none of the rules produces any change:

add-prop If a transition into state s is labeled with $x := c$ then add the binding $< x, c >$ to $s{\uparrow}\mathcal{SH}$.[14]

split-state If state s contains bindings $< x, c^1 >, \ldots, < x, c^k >$ $(k > 1)$ then replace s by k propositionally equivalent[15] states $s^1, \ldots, s^k$, where s^ℓ contains $< x, c^\ell >$ and all bindings of s not involving x ($\ell \in [1 : k]$). Each s^ℓ has the same outgoing transitions as s, but has as incoming transitions only the incoming transitions of s that are consistent[16] with $x = c^\ell$.

propagate-value If state s contains $< x, c >$ and there exists a transition from s to s' not labeled with an assignment to x, then add $< x, c >$ to s'.

The function of the above three rules is to resolve any inconsistencies that arise due to the replication of multiple assignments by creating new global states as needed. The three rules above cannot introduce any new cycles, nor any states that are not propositionally equivalent to a state already occurring in M. Also, termination is guaranteed since the number of possible states and/or transitions is finite; hence, eventually no new states and/or transitions can be added. Finally, we let the set of initial states of M', namely S'_0, be

$$\{s \mid s \text{ is a state of } M' \text{ and } s \text{ is propositionally equivalent to some state in } S_0\}.$$

PROPOSITION 4.1.2.1. *Let f be an arbitrary formula of* CTL. *Then*

$$M, S_0 \models f \textit{ iff } M', S'_0 \models f.$$

PROOF. It is straightforward to establish a bisimulation [Clarke et al. 1986] between M and M': every state in M is bisimilar to all states M' that are propositionally equivalent to it. Since the only rule that adds new states is **split-state**, and since this rule preserves all outgoing transitions, i.e., all successor states, it follows (by a simple induction) that propositionally equivalent states are indeed bisimilar. Theorem 2 of Clarke et al. [1986] is as follows:

[14] Note the abuse of notation here. Technically, $s{\uparrow}\mathcal{SH}$ is a mapping; it relates each $x \in \mathcal{SH}$ to exactly one value in D_x. However, in the middle of applying the rules, $s{\uparrow}\mathcal{SH}$ could relate some $x \in \mathcal{SH}$ to more than one value in D_x. The **split-state** rule ensures that upon termination $s{\uparrow}\mathcal{SH}$ is a mapping, for all states s in M. We use the notation $< x, c >$ to denote a variable to value binding here. We say that state s contains the binding $< x, c >$ iff $< x, c > \in s{\uparrow}\mathcal{SH}$.

[15] Two states are propositionally equivalent iff they agree on all atomic propositions.

[16] An incoming transition is consistent with $x = c^\ell$ iff either (1) it is labeled with $x := c^\ell$, or (2) it is not labeled with an assignment (or equivalently, it is labeled with *skip*), and it originates in a state containing the binding $< x, c^\ell >$.

> Let M^1 and M^2 be two structures that correspond.
> Then for all formulae f of CTL*,
> $M^1, s_0^1 \models f$ iff $M^2, s_0^2 \models f$.

Here the term "correspond" means "are bisimilar." Although this theorem is given for structures M_1, M_2 with single initial states s_0^1, s_0^2 respectively, it is easily seen to generalize to structures with a set of initial states. Furthermore, the logic CTL* [Emerson 1990] subsumes CTL. Thus, applying the theorem to M, M', we conclude: $M, S_0 \models f$ iff $M', S'_0 \models f$ for any formula f of CTL. $\square$

4.1.3 *Step 1.3: Extract a Correct High-Atomicity Program.* Next, we extract a correct high-atomicity program $P = P_1 \| \cdots \| P_K$ from M' using the program extraction definition (2.3). We mention our convention, which we use in all figures of Kripke structures (synchronization skeletons), of showing in a global (local) state only the atomic propositions that are *true* in that global (local) state. The atomic propositions not shown can be taken to be *false* in the global (local) state. Note also, that in all figures of Kripke structures, we show only the reachable portions of the actual structures.

4.1.4 *Step 1.4: Simplify the Guards.* The program extraction definition (2.3) produces guards that are in disjunctive normal form, where each disjunct contributes exactly one transition. Each guard, in principle, may read the entire global state, i.e., the atomic propositions of all other processes, and the values of all shared variables. Often, it is not necessary to read all the components of the global state, and the guards can be "simplified" considerably.

Consider an arbitrary arc $(s_i, B \to A, t_i)$ of P_i. The guard B is tested only in global states whose P_i-component is s_i. Hence, B can be replaced by any B' that has the same value as B in all such global states, and possibly differs from B in other global states. Hence, any B' such that

$$\text{for all } s \in S : s{\uparrow}i = s_i \text{ implies } s \models (B \equiv B') \qquad \text{(GS)}$$

can be used as a guard instead of B, i.e., the arc $(s_i, B \to A, t_i)$ is replaced by $(s_i, B' \to A, t_i)$.

Two special cases of (GS) that are useful in practice are as follows:

Guard elimination: $B' = $ *true*. If we establish

$$\text{for all } s \in S : s{\uparrow}i = s_i \text{ implies } s \models B$$

then B can be replaced by *true*, i.e., the arc is enabled whenever "local control" is at its start state.

Conjunct elimination: $B = B'' \vee (b_1 \wedge \ \ldots\ \wedge b_n)$, and $B' = B'' \vee (b_1 \wedge \ \ldots\ \wedge b_{n-1})$. If we establish

$$\text{for all } s \in S : s{\uparrow}i = s_i \text{ implies } s \models [(b_1 \wedge \ \ldots\ \wedge b_{n-1}) \Rightarrow b_n]$$

then B can be replaced by B', since we would have:

$$\text{for all } s \in S : s{\uparrow}i = s_i \text{ implies } s \models [(b_1 \wedge \ \ldots\ \wedge b_{n-1}) \equiv (b_1 \wedge \ \ldots\ \wedge b_n)]$$

and so

for all $s \in S : s{\uparrow}i = s_i$ implies $s \models [B'' \vee (b_1 \wedge \ \ldots\ \wedge b_{n-1}) \ \equiv\ B'' \vee (b_1 \wedge \ \ldots\ \wedge b_n)]$.

Since the program extraction definition (2.3), produces guards in disjunctive normal form, this simplification method should be quite useful.

4.2 Phase 2: Decompose the Initial Program

4.2.1 *Step 2.1: Introduce the Externally Visible Location Counters.* We now introduce the externally visible location counter L_i, and replace all references to the atomic propositions in $\mathcal{AP}_i$ by references to L_i:

—Every occurrence of an atomic proposition $Q_i \in \mathcal{AP}_i$ in some guard B (occurring in the skeleton for some process P_j, $j \neq i$) is replaced by "$Q_i \in L_i$."

—Every arc $(s_i, B \rightarrow A, t_i)$ of P_i, such that $\mathcal{L}[s_i] \neq \mathcal{L}[t_i]$, is replaced by the arc $(s_i, B \rightarrow A // L_i := \mathcal{L}[t_i], t_i)$.

The first transformation replaces a read of an atomic proposition in $\mathcal{AP}_i$ by a read of L_i. The second transformation ensures that L_i implements the atomic propositions in $\mathcal{AP}_i$ correctly; in other words, it ensures that LOC (Section 2.2) holds for every i-state in P_i. We extend the definition of global state to provide for a value assigned to L_i by a global state s: $s(L_i) = s{\uparrow}i(L_i)$. Finally, we mention that the state-to-formula definition (2.1) should be modified (to take into account the introduction of the L_i) as follows: every conjunct of the form Q_i, $(\neg Q_i)$ is replaced by $Q_i \in L_i$ $(Q_i \notin L_i)$ respectively.

Since $\mathcal{L}[t_i]$ is constant for a given local state t_i, the assignment $L_i := \mathcal{L}[t_i]$ can be implemented by a single atomic write operation. Thus, the assignment operation A of an arc $(s_i, B \rightarrow A, t_i)$ has the form $//_{m \in [1:n]} A^m$, $(n \geq 0)$, where each A^m has either the form $x := c$ $(x \in \mathcal{SH}, c \in D_x)$, or the form $L_i := \mathcal{L}[t_i]$, with at most one A^m having the latter form. If $\mathcal{L}[s_i] = \mathcal{L}[t_i]$, then no A^m has the form $L_i := \mathcal{L}[t_i]$, and if $\mathcal{L}[s_i] \neq \mathcal{L}[t_i]$, then exactly one A^m has this form.

4.2.2 *Step 2.2: Decompose the Test-and-Set Arcs.* Our next step is to decompose every test-and-set arc in P into a guarded and nonwriting arc (for the "test"), followed by an unguarded and writing arc (for the "set"):

```
—For every test-and-set arc $AR = (s_i, B \rightarrow A, t_i)$
        replace $AR$ by $AR'$ and $AR''$, where
                $AR' = (s_i, B \rightarrow skip, u_i)$
                $AR'' = (u_i, true \rightarrow A, t_i)$
```

where u_i is a "new" i-state (i.e., it does not already occur in P_i), and $\mathcal{L}[u_i]$ is set to $\mathcal{L}[s_i]$.

4.2.3 *Step 2.3: Decompose the Multiple-Writing Arcs and Number the Local States.* Finally, we replace every unguarded and multiple-writing arc by a set of sequences of unguarded and single-writing arcs, where each sequence represents one order of serialization of the write operations of the original multiple-writing arc:

```
—For every multiple-writing arc $AR = (u_i, true \rightarrow //_{m \in [1:n]} A^m, t_i)$
        remove $AR$ from $P$;
        for every permutation $m_1, \ldots, m_n$ of $1, \ldots, n$,
```

add $AR^{m_1}, \ldots, AR^{m_n}$ to P, where

$$\begin{aligned} AR^{m_1} &= (u_i, \mathit{true} \rightarrow A^{m_1}, u_i^{m_1}) \\ AR^{m_k} &= (u_i^{m_{k-1}}, \mathit{true} \rightarrow A^{m_k}, u_i^{m_k}) \text{ for } k \in [2 : n-1] \\ AR^{m_n} &= (u_i^{m_{n-1}}, \mathit{true} \rightarrow A^{m_n}, t_i) \end{aligned}$$

where the i-states $u_i^{m_1}, \ldots, u_i^{m_{n-1}}$ do not previously occur in P (for every permutation), i.e., we use "new" local states for each permutation.

Also, the local atomic proposition valuations of the new local states are given as follows:

—If $//_{m\in[1:n]}A^m$ does not assign to L_i, (and so no A^{m_k} has the form $L_i := \mathcal{L}[t_i]$), then set $\mathcal{L}[u_i^{m_k}]$ to $\mathcal{L}[s_i]$ for all k in $[1 : n-1]$.

—If $//_{m\in[1:n]}A^m$ assigns to L_i, then exactly one A^{m_ℓ}, $\ell \in [1 : n]$ has the form $L_i := \mathcal{L}[t_i]$. In this case:
set $\mathcal{L}[u_i^{m_k}]$ to $\mathcal{L}[s_i]$ for all k in $[1 : \ell - 1]$
set $\mathcal{L}[u_i^{m_k}]$ to $\mathcal{L}[t_i]$ for all k in $[\ell : n-1]$

Call the resulting program $P' = P_1' \| \cdots \| P_K'$. Looking at Figure 13, we see that it contains propositionally equivalent but distinct local states. We now assign numbers to local states (see Section 3.2) so that all propositionally equivalent but distinct local states in P' are distinguished from each other by their numbers (i.e., so that (NUM)—see Section 3.2—holds).

Let $P'' = P_1'' \| \cdots \| P_K''$ be the program that results from the local state numbering. We can now state the following proposition. To facilitate its proof, we shall make the following assumption:

Technical Assumption: *P contains no arcs that are both unguarded and nonwriting.*

If P does contain such arcs, we can always convert them to single-writing arcs by adding a new atomic proposition which is "written to" by such arcs, but is never "read" by any arc, and so does not change P's behavior.

PROPOSITION 4.2.3.1. *Every arc in the skeletons of P'' is either guarded and nonwriting, or unguarded and single-writing.*

PROOF. By construction of Phase 2, P'' contains no test-and-set arcs. Hence by the syntactic arc attributes definition (3.1.2), every arc in P'' is nonwriting or unguarded. By this and the technical assumption (given immediately above), every arc in P'' is guarded and nonwriting or unguarded and writing. Finally, by construction of Phase 2, P'' contains no multiple-writing arcs. Hence every arc in P'' is guarded and nonwriting or unguarded and single-writing. □

4.3 Phase 3: Generate the Global-State Transition Diagram

In Phase 3, we generate the global-state transition diagram $M'' = (S_0', S, R'')$ of P'' by applying the global-state transition diagram definition (2.2.2) to P''.[17]

[17] S_0' is the set of initial states of the structure M' derived in Phase 1. See Section 4.1.2.

For each conjunct f of the specification, except those of the form $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$, label the states of M'' according to the following labeling rules, where $p, q, r, h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, and $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$.

—If $f = h$, then label every initial state in M'' (i.e., every state in S'_0) with h.

—If $f = \mathsf{AG}h$, then label every reachable state in M'' with h.

—If $f = \mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$ then label every reachable state s in M'' such that $s \models p$ with $\mathsf{A}[q\mathsf{U}r]$.

—If $f = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$ then label every reachable state s in M'' such that $s \models p_i$ with $\mathsf{EX}_i(p_i \vee q_i)$.

Fig. 5. The labeling rules.

Our use of local-state numbers makes it clear which local states a global state projects onto. For example, the state $[T_1^2\ 2\ N_2^1]$ in Figure 15 has a P_1-projection of $[T_1^2]$, i.e., the 1-state indicated by T_1^2 in Figure 14. Since there are two other propositionally equivalent P_1-states in Figure 14, namely those labeled with T_1^1, T_1^3 respectively, it would be impossible to distinguish these P_1-states from the P_1-state labeled with T_1^2 were it not for the use of local-state numbers.

4.4 Phase 4: Delete Portions of the Global-State Transition Diagram that Violate the Specification

The global-state transition diagram generated in Phase 3 will not, in general, satisfy the CTL specification. This is because the syntactic decomposition of the skeleton arcs performed in Phase 2 introduces the possibility of new interleavings, leading to computation paths that generate previously unreachable states. For example, suppose that the arcs AR_i, AR_j in processes P_i, P_j of $P = P_1 \| \cdots \| P_K$ are decomposed into arcs AR_i^1, AR_i^2 and AR_j^1, AR_j^2 respectively, in the processes P''_i, P''_j of the decomposed program $P'' = P''_1 \| \cdots \| P''_K$ produced by Phase 2. In the execution of P, either AR_i is executed before AR_j or vice versa. In the execution of P'', it is possible for the execution of AR_i^1, AR_i^2 and AR_j^1, AR_j^2 to be interleaved, e.g., $AR_i^1; AR_j^1; AR_i^2, AR_j^2$. This represents a new behavior, not possible in program P. Such new behaviors can lead to global states (or cycles of global states) that were previously unreachable. These new states/cycles could violate the specification. For example, in Figure 15, the dashed path results from such a new behavior, and leads to the state $[C_1^1\,2\,C_2^1]$, which violates the conjunct $\mathsf{AG}(\neg(C_1 \wedge C_2))$ of the mutual exclusion specification.

Our solution to this problem is to delete all portions of the global-state transition diagram that violate the specification. Provided that not all initial states are deleted as a result, the remaining structure will then satisfy the specification, and a correct atomic read/write program could be extracted from it. We proceed as follows.

4.4.1 *Step 4.1: Label the Reachable States.* First, we *label* every reachable state of M'' with a set of CTL formulae. The label of each state (notated as $label(s)$) contains exactly the CTL formulae that must be satisfied by that state in order that every initial state of M'' satisfy the specification. The labeling procedure is given in Figure 5.

4.4.2 *Step 4.2: Prune the Structure So that the Labels Are Satisfied.* Next, we check that every state of M'' satisfies all the formulae in its label. If some state of

```
procedure $delete(TR_i)$

Let $\mathcal{F}_i$ be the family in $M''$ containing $TR_i$, and let $AR_i$ be the
skeleton arc corresponding to $\mathcal{F}_i$.
if $\mathcal{F}_i$ is guarded and nonwriting then
        remove $TR_i$ from $M''$;
        if $\mathcal{F}_i$ is now empty, then remove $AR_i$ from $P''_i$
else ($\mathcal{F}_i$ is unguarded and single-writing)
        remove all transitions in $\mathcal{F}_i$ from $M''$;
        remove $AR_i$ from $P''_i$
endif;
recompute the "deletable" attribute for all arcs of $P''_i$
```

Fig. 6. The delete transition procedure.

Prop-rule $h \in label(s)$ and $s \not\models h$.
If $s \in S'_0$, then delete s. Otherwise, make s unreachable in M'', i.e., find one deletable transition TR_i from every initialized path ending in s, and remove TR_i from M'' by invoking $delete(TR_i)$ (an initialized path is a path starting in an initial state).

AU-rule $\mathsf{A}[q\mathsf{U}r] \in label(s)$ and $s \not\models \mathsf{A}[q\mathsf{U}r]$.
Find one deletable transition TR_i from every fullpath π starting in s such that $\pi \not\models [q\mathsf{U}r]$, and remove TR_i from M'' by invoking $delete(TR_i)$.

EX_i-rule $\mathsf{EX}_i(p_i \vee q_i) \in label(s)$ and $s \not\models \mathsf{EX}_i(p_i \vee q_i)$.
If $s \in S'_0$, then delete s. Otherwise, make s unreachable in M'', as in the **Prop-rule**.

EX-rule $s \not\models \mathsf{EX}true$, i.e., s has no successors.
If $s \in S'_0$, then delete s. Otherwise, make s unreachable in M'', as in the **Prop-rule**.

Arc-rule $(s_i, B \rightarrow A, t_i)$ is an arc in P''_i such that either (1) P''_i contains no arc with start state t_i, or (2) P''_i contains no arc with finish state s_i, and $s_i \notin S'_0{\uparrow}i$[18] (i.e., s_i is not an initial local state).

Remove $(s_i, B \rightarrow A, t_i)$ from P''_i, and its corresponding family from M''.

The name and activation condition (for a particular reachable state s) of each rule is given first, with the action required by the rule given on succeeding lines.
For all the above rules, whenever a state s in S'_0 is deleted, all transitions in R'' which involve s (as either a begin or end state) are also deleted. If any of these transitions are undeletable, then the synthesis method terminates with failure.

Fig. 7. The deletion rules.

M'' does not satisfy a formula in its label, then M'' does not satisfy the specification and must be modified. We modify M'' by deleting initial states and transitions. Now deleting a single transition TR_i (of some process P''_i of P'') may cause the arc AR_i corresponding to the family containing TR_i to become guarded. If AR_i was previously single-writing, then AR_i now becomes test-and-set. We avoid this possibility by deleting all transitions of the family containing TR_i, thereby deleting AR_i entirely. However, this may leave P''_i incapable of infinite behavior, e.g., if P''_i was previously a single "cycle." We say that an arc AR_i of P''_i is *deletable* iff its deletion leaves at least one cycle in P''_i, i.e., leaves P''_i capable of infinite behavior. Otherwise we say that AR_i is *nondeletable.* We only delete transitions in M'' that correspond to deletable arcs. A transition TR_i is deleted by invoking the procedure $delete(TR_i)$, shown in Figure 6.

The actual deletions to be carried out are given by a set of deletion rules, shown in Figure 7. The deletion rules are applied as long as possible. Since M'' is finite, and each application of a deletion rule results in the deletion of at least one state or one transition in M'', we eventually terminate, i.e., reach a situation in which none of the deletion rules are applicable. Upon termination, let S_0''', R''' be the set of undeleted initial states, undeleted and reachable (from S_0''') transitions, respectively. If S_0''' is empty, then all the initial states have been deleted, and we are therefore unable to extract a program from M'' that satisfies the problem specification. In this case, our synthesis method terminates with failure. This possibility of termination with failure means that our synthesis method is not *complete*, i.e., it may not always produce an atomic read/write program satisfying a given specification, even if such a program does in fact exist. The method is sound however, as we subsequently establish. If S_0''' is nonempty, let M''' be the structure (S_0''', S, R'''). We show that M''' satisfies the CTL specification.

PROPOSITION 4.4.2.1 (SOUNDNESS). *If $S_0''' \neq \emptyset$, and f is a conjunct of the specification, then*

$$M''', S_0''' \models f^*$$

where $f^* = f$ iff has one of the forms h, $\mathsf{AG}h$, $\mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$, and $f^* = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i(p_i \vee q_i))$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$ if $f = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$ respectively, and where $p, q, r, h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$.

PROPOSITION 4.4.2.2. *Every family in M''' is either guarded and nonwriting or unguarded and single-writing.*

PROOF. By Proposition 4.2.3.1, every arc in the skeletons of P'' is either guarded and nonwriting, or unguarded and single-writing. Now M'' is generated by applying the global-state transition diagram definition (2.2.2) to P''. From the global-state transition diagram definition (2.2.2) and the unguarded family definition (3.3.2), we see that an unguarded arc generates an unguarded family. From the global-state transition diagram definition (2.2.2), and the family write attributes definition (3.3.1), a nonwriting (single-writing) arc generates a nonwriting (single-writing) family, respectively.

Hence every family in M'' is either guarded and nonwriting or unguarded and single-writing. By construction of Phase 4, every unguarded family in M'' either survives intact, i.e., none of its transitions are deleted, and hence it is unguarded (and single-writing) in M''', or is deleted entirely. Thus the only guarded families in M''' are (subsets of) those that were guarded in M''. Thus, by the family write attributes definition (3.3.1) these families are nonwriting in M''', since they are nonwriting in M''. Since every family in M''' corresponds to a family in M'', the proposition follows. □

4.5 Phase 5: Extract a Correct Atomic Read/Write Program

4.5.1 *Step 5.1: Ensure Atomic Read/Write Form.* The Kripke structure produced by Phase 4 is not, in general, guaranteed to be in atomic read/write form. What is missing is that the guarded and nonwriting families must be single-reading and nonwriting. Boolean function CHECK-STRUCTURE(M) given in Figure 8 tests all guarded and nonwriting families in a Kripke structure M, and returns true

```
**Boolean Function** CHECK-STRUCTURE($M$)
/* input: Kripke structure $M$ such that every family in $M$ is guarded and nonwriting or
          unguarded and single-writing
   output: true — if $M$ is an atomic read/write Kripke structure, i.e.,
                  every guarded and nonwriting family in $M$ is single-reading
           false — otherwise
*/
**for each** guarded and nonwriting family $\mathcal{F}$ of $M$ **do**
    **if** $\neg$CHECK-FAMILY($M, \mathcal{F}$) **then return**($false$) **endif**
    /* If some guarded and nonwriting family $\mathcal{F}$ is not single-reading, then $M$ is not in
       atomic read/write form */
**endfor**;
**return**($true$)
```

```
**Boolean Function** CHECK-FAMILY($M, \mathcal{F}$)
/* input: Kripke structure $M$ and family $\mathcal{F}$ of $M$
   output: true — if $\mathcal{F}$ is single-reading (in this case, $\mathcal{F}.srguard$ gives a suitable guard for the
                  corresponding extracted arc)
           false — otherwise
*/
$simple := \{``Q_i \in L_i", ``Q_i \notin L_i" \mid Q_i \in \mathcal{AP}_i, i \in [1:K]\} \cup \{``x = c" \mid x \in \mathcal{SH} \wedge c \in D_x\}$;
/* $simple$ contains all possible simple terms */
$ok := \emptyset$;
**for each** $b \in simple$ **do**
    **if** OK($M, \mathcal{F}, b$) **then** $ok := ok \cup \{b\}$ **endif**
    /* Simple terms that generate only transitions in $\mathcal{F}$ are added to $ok$.
       This ensures that condition G2 of Definition 3.3.3 is satisfied. */
**endfor**;
**if** COV($\mathcal{F}, ok$) **then**
    $\mathcal{F}.srguard := ``\bigvee_{b \in ok} b"$;
    **return**($true$)
    /* If all the simple terms in $ok$ collectively generate all the transitions in $\mathcal{F}$, then declare
    $\mathcal{F}$ to be single-reading. This ensures that condition G1 of Definition 3.3.3 is satisfied. */
**else**
    **return**($false$)
**endif**
```

where the predicates OK($M, \mathcal{F}, b$), COV($\mathcal{F}, ok$) are as follows:

```
OK($M, \mathcal{F}, b$) $\equiv$ $\forall$ reachable states $s$ of $M$ $(s(b) = true \Rightarrow \exists T \in \mathcal{F}\,(T.begin = s))$
        /* $b$ is true only in states from which there is an outgoing transition of $\mathcal{F}$ */
COV($\mathcal{F}, ok$) $\equiv$ $\forall T \in \mathcal{F}\,(\exists b \in ok\,(T.begin(b) = true))$
        /* The simple terms in $ok$ collectively generate all the transitions of $\mathcal{F}$ */
```

Fig. 8. Function CHECK-STRUCTURE(M) for testing the atomic read/write form of a Kripke Structure M.

iff they are all single-reading as well. We invoke CHECK-STRUCTURE(M'''). If this returns *true*, then, by Proposition 4.4.2.2, we can conclude that every family in M''' is either single-reading and nonwriting, or unguarded and single-writing, i.e., that M''' is an atomic read/write Kripke structure. Given that M''' is in atomic read/write form, the atomic read/write program extraction definition (3.4.1) can be applied to extract the final atomic read/write program. If, however, M''' is not in atomic read/write form, then there are one or more guarded and nonwriting families that do not satisfy (at least one of) the conditions (G1), (G2) of the single-reading family definition (3.3.3). For each such family, we attempt to make (G1) and (G2) true by deleting reachable states that violate (G1) or (G2) (or both). Since such deletions may, in general, cause violation of the specification, we must repeat Phase 4 after one or more of these deletions are performed.

CHECK-STRUCTURE(M) works by invoking CHECK-FAMILY($M, \mathcal{F}$) for each guarded and nonwriting family $\mathcal{F}$ in M. CHECK-FAMILY($M, \mathcal{F}$) returns *true* if and only if $\mathcal{F}$ is single-reading, and it works as follows. Recall that a simple term is a formula of the form $Q_i \in L_i$ or $x = c$. Because a simple term refers to only one variable (either L_i or x), its value can be checked with a single atomic read operation. Recall also that the truth of a disjunction of simple terms can be *verified* by a single atomic read operation, namely the read of the variable referenced by a simple term disjunct that happens to be true. CHECK-FAMILY($M, \mathcal{F}$) now attempts to satisfy Definition 3.3.3 by finding a suitable "covering" $\mathcal{F}_1, \ldots, \mathcal{F}_n$ of $\mathcal{F}$ (i.e., subsets $\mathcal{F}_1, \ldots, \mathcal{F}_n$ of $\mathcal{F}$ such that $\mathcal{F} = \bigcup_{k \in [1:n]} \mathcal{F}_k$) and simple terms $b_1, \ldots, b_n$ such that the transitions in each $\mathcal{F}_k$ can be "generated" by including b_k as a disjunct of the guard of $\mathcal{F}$. CHECK-FAMILY($M, \mathcal{F}$) starts with the set of all possible simple terms (this is of size $O(|M|)$). If a simple term generates only transitions in $\mathcal{F}$ then it is added to the set *ok* (see the definition of OK(b) in Figure 8). If, after all simple terms have been examined, the disjunction of the terms in *ok* (call them $b_1, \ldots, b_n$) generates all the transitions in $\mathcal{F}$ (see the definition of COV($\mathcal{F}, ok$) in Figure 8) then $b_1, \ldots, b_k$, together with each $\mathcal{F}_k$ generated by b_k, satisfy Definition 3.3.3. In this case, CHECK-FAMILY($M, \mathcal{F}$) returns *true*. Otherwise, it returns *false*.

4.5.2 *Step 5.2: Extract the Atomic Read/Write Program.* Once M''' is in atomic read/write form, we apply the atomic read/write program extraction definition (3.4.1) to extract the final atomic read/write program $P''' = P_1''' \| \cdots \| P_K'''$.

We now establish the soundness of the synthesis method.

Proposition 4.5.2.1. *Let $M^{iv} = (S_0''', S, R^{iv})$ result from applying the global-state transition diagram definition (2.2.2) to P'''. If $S_0''' \neq \emptyset$, and f is a conjunct of the specification, then*

$$M^{iv}, S_0''' \models f^*$$

where $f^* = f$ if f has one of the forms h, $\mathsf{AG}h$, $\mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$, and $f^* = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i(p_i \vee q_i))$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$ if $f = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$ respectively, and where $p, q, r, h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$.

Proof. Assume that $S_0''' \neq \emptyset$ and that f is a conjunct of the specification. By Proposition 4.4.2.1, we have $M''', S_0''' \models f^*$. Hence, by Proposition 3.4.4, we conclude $M^{iv}, S_0''' \models f^*$. □

Theorem 4.5.2.2 (Soundness for Synchronization Skeleton Model). *If $S_0''' \neq \emptyset$ and f is a conjunct of the specification, then P''' is an atomic read/write program that satisfies f^*, where $f^* = f$ if f has one of the forms h, $\mathsf{AG}h$, $\mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$, and $f^* = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i(p_i \vee q_i))$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$ if $f = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$ respectively, and where $p, q, r, h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$.*

PROOF. Assume $S_0''' \neq \emptyset$. By construction of Phase 5, M''' is an atomic read/write Kripke structure. Since P''' results from applying the atomic read/write program extraction definition (3.4.1) to M''' (by construction of Phase 5), we conclude by Proposition 3.4.2 that P''' is an atomic read/write program.

Finally, since M^{iv} is the global-state transition diagram of P''', we conclude from Proposition 4.5.2.1 and the definition of correctness property (see Section 2.5) that P''' satisfies f^*. □

4.6 Summary of the Method

To give an overview of our method, we summarize the steps as follows. For some steps, we indicate the Kripke structure or program that is produced (or modified) by that step.

—Phase 1: Synthesize a Correct High-Atomicity Program
  - Step 1.1: Derive (or Specify) the Initial Kripke Structure M
  - Step 1.2: Replicate Multiple Assignments (M')
  - Step 1.3: Extract a Correct High-Atomicity Program (P)
  - Step 1.4: Simplify the Guards

—Phase 2: Decompose the High-Atomicity Program
  - Step 2.1: Introduce the Externally Visible Location Counters
  - Step 2.2: Decompose the Test-and-Set Arcs
  - Step 2.3: Decompose the Multiple-Writing Arcs (P') and Number the Local States (P'')

—Phase 3: Generate the Global-State Transition Diagram (M'')

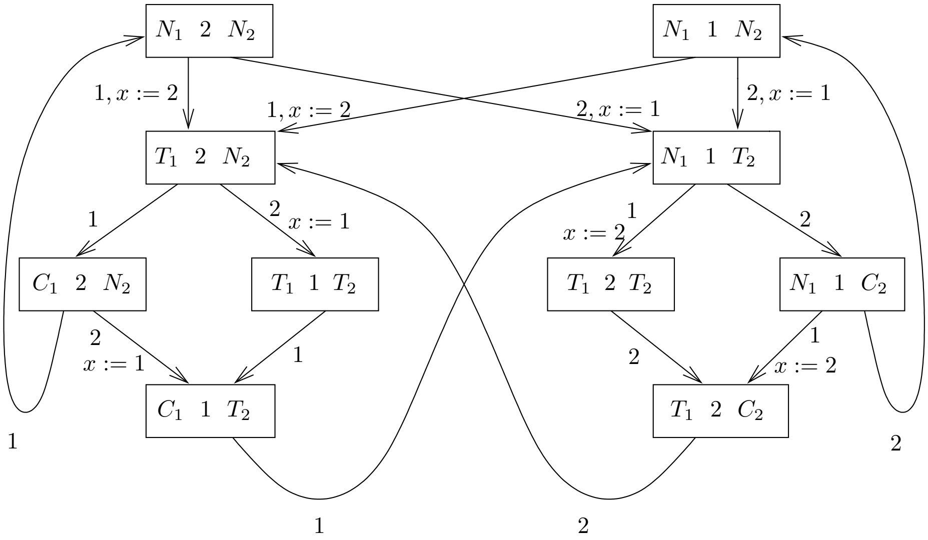
—Phase 4: Delete Portions of the Global-State Transition Diagram that Violate the Specification
  - Step 4.1: Label the Reachable States
  - Step 4.2: Prune Structure So That Labels Are Satisfied (M''')

—Phase 5: Extract a Correct Atomic Read/Write Program
  - Step 5.1: Ensure Atomic Read/Write Form (M''')
  - Step 5.2: Extract the Atomic Read/Write Program (P''')

5. EXTENDED EXAMPLE: TWO-PROCESS MUTUAL EXCLUSION

We now present a detailed example of the use of our method to synthesize an atomic read/write solution to the two-process mutual exclusion problem. We show the working of each phase of our method, along with the intermediate results.

We note that whenever the atomic propositions in $\mathcal{AP}_i$ are mutually exclusive and exhaustive (in other words, every atomic proposition is true in exactly one i-state, with no two atomic propositions being true in the same i-state), then we can encode the externally visible location counter L_i efficiently by setting its value to the name of the proposition that is currently true, rather than the singleton set containing that proposition. That is, if $Q_i \in \mathcal{L}[s_i]$ then set $s_i(L_i)$ to "Q_i" rather than to {"Q_i"}.



The initial state set is { [N_1 2 N_2], [N_1 1 N_2] }

Fig. 9. Global-state transition diagram for the two-process mutual exclusion problem after application of the propagation rules in Section 4.1.2 to Figure 2.

5.1 Phase 1

We discuss each step in turn.

Step 1.1: *Derive (or Specify) the Initial Kripke Structure.* We start with the two-process mutual exclusion specification given in Section 2.4, and apply the CTL decision procedure of Emerson and Clarke [1982] to it. Figure 2 shows the resulting model.

Step 1.2: *Replicate Multiple Assignments.* This step replicates the assignments $x := 2$, $x := 1$ in Figure 2. Figure 9 shows the resulting Kripke structure. The assignment $x := 2$ which, in Figure 2 is executed only in the P_1-transition $[N_1\ T_2] \xrightarrow{1,x:=2} [T_1\ x = 2\ T_2]$, is, in Figure 9, executed in every P_1-transition starting in a state whose P_1-component is N_1 and ending in a state whose P_1-component is T_1, i.e., the transitions $[N_1\ 2\ N_2] \xrightarrow{1,x:=2} [T_1\ 2\ N_2]$, $[N_1\ 1\ N_2] \xrightarrow{1,x:=2} [T_1\ 2\ N_2]$, $[N_1\ 1\ T_2] \xrightarrow{1,x:=2} [T_1\ 2\ T_2]$, and $[N_1\ 1\ C_2] \xrightarrow{1,x:=2} [T_1\ 2\ C_2]$. $x := 1$ is similarly replicated. Another point is that every state in Figure 9 assigns a value to x, whereas some states of Figure 2 do not (note our convention in Figure 9, which we use henceforth, of just writing down the value of x, rather than "$x =$ value," as in Figure 2). Thus we see that every state (except $[T_1\ x = 2\ T_2]$ and $[T_1\ x = 1\ T_2]$) in Figure 2 potentially represents two states—one for each element of the domain $\{1, 2\}$ of x. For example, in Figure 9, $[N_1\ N_2]$ has been split into two initial states, $[N_1\ 1\ N_2]$ and $[N_1\ 2\ N_2]$.

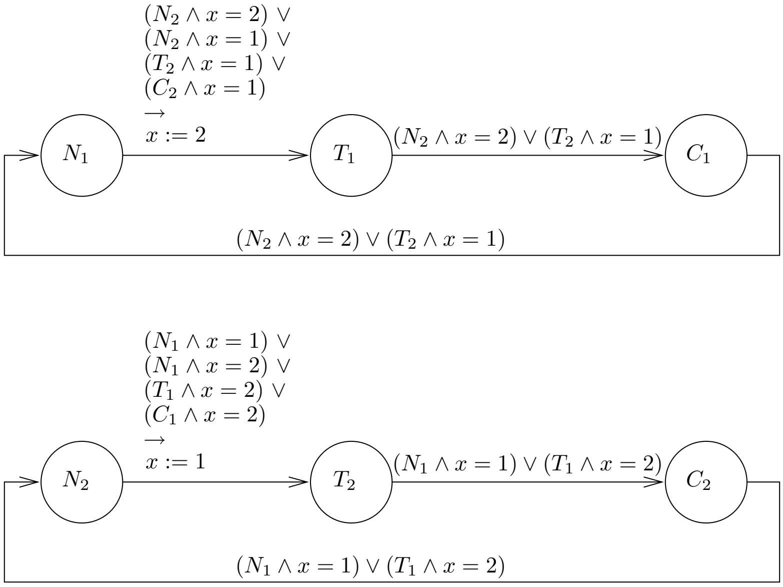

$(N_2 \wedge x = 2) \vee$
$(N_2 \wedge x = 1) \vee$
$(T_2 \wedge x = 1) \vee$
$(C_2 \wedge x = 1)$
$\rightarrow$
$x := 2$
N_1
T_1
$(N_2 \wedge x = 2) \vee (T_2 \wedge x = 1)$
C_1
$(N_2 \wedge x = 2) \vee (T_2 \wedge x = 1)$
$(N_1 \wedge x = 1) \vee$
$(N_1 \wedge x = 2) \vee$
$(T_1 \wedge x = 2) \vee$
$(C_1 \wedge x = 2)$
$\rightarrow$
$x := 1$
N_2
T_2
$(N_1 \wedge x = 1) \vee (T_1 \wedge x = 2)$
C_2
$(N_1 \wedge x = 1) \vee (T_1 \wedge x = 2)$

Fig. 10. Program for the two-process mutual exclusion problem extracted from the Kripke structure of Figure 9.

Step 1.3: *Extract a Correct High-Atomicity Program.* Figure 10 shows the program extracted from the Kripke structure of Figure 9, using Definition 2.3.

Step 1.4: *Simplify the Guards.* Figure 11 illustrates the synchronization skeletons that result when the guards in Figure 10 are simplified. The guard $(N_2 \wedge x = 2) \vee (T_2 \wedge x = 1)$ in P_1 (from T_1 to C_1) has been simplified to $N_2 \vee x = 1$, using the conjunct elimination method of step 1.4. For example, from Figure 9, we see that the condition for replacing $(N_2 \wedge x = 2) \vee (T_2 \wedge x = 1)$ by $N_2 \vee (T_2 \wedge x = 1)$ is

$$\begin{array}{l}[T_1\ x = 2\ N_2] \models (N_2 \Rightarrow x = 2) \ \text{ and}\\ [T_1\ x = 1\ T_2] \models (N_2 \Rightarrow x = 2) \ \text{ and}\\ [T_1\ x = 2\ T_2] \models (N_2 \Rightarrow x = 2) \ \text{ and}\\ [T_1\ x = 2\ C_2] \models (N_2 \Rightarrow x = 2).\end{array}$$

This is easily verified. In a similar manner, we can then replace $N_2 \vee (T_2 \wedge x = 1)$ by $N_2 \vee x = 1$.

Using the guard elimination method of step 1.4, the guard $(N_2 \wedge x = 2) \vee (N_2 \wedge x = 1) \vee (T_2 \wedge x = 1) \vee (C_2 \wedge x = 1)$ in P_1 in Figure 10 has been replaced by *true*. Likewise, the guard $(N_2 \wedge x = 2) \vee (T_2 \wedge x = 1)$ in P_1 (from C_1 to N_1) has been simplified to *true*. The program resulting from all of these guard simplifications (applied symmetrically to P_2 as well) is shown in Figure 11.

5.2 Phase 2

We discuss each step in turn.

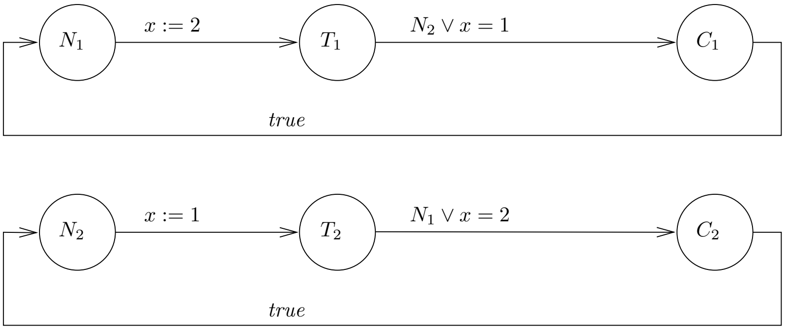


Fig. 11. Two-process mutual exclusion program after guard simplification.

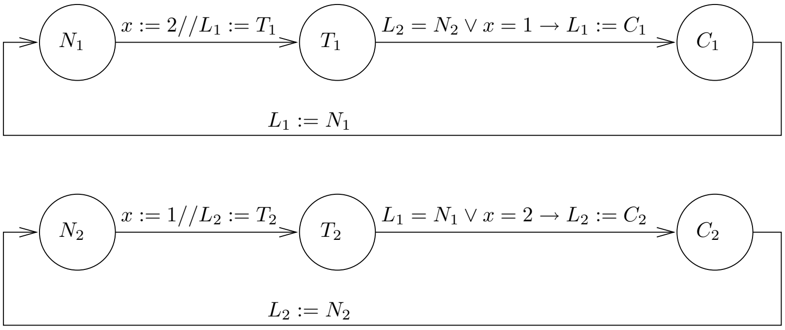


Fig. 12. Two-process mutual exclusion program after introduction of the externally visible location counters.

Step 2.1: Introduce the Externally Visible Location Counters. Figure 12 shows the program of Figure 11 after the externally visible location counters L_1 and L_2 have been introduced. Note that we have used the convention (mentioned at the beginning of Section 5) of writing $L_i =$ "Q_i" instead of $Q_i \in L_i$ when $s_i(L_i)$ assigns *true* to exactly one proposition. This convention is used in the sequel for the mutual exclusion example (but we abuse notation slightly by omitting the quotation marks around the atomic proposition names).

Step 2.2: Decompose the Test-and-Set Arcs, and Step 2.3: Decompose the Multiple-Writing Arcs and Number the Local States. Figure 13 shows the program of Figure 12 after all arcs have been decomposed. Note that the arc labeled $x := 2 // L_1 := T_1$ has been decomposed into two sequences, each sequence corresponding to one of the two possible serializations of the two write operations $x := 2$ and $L_1 := T_1$.

In Figure 13, there are three local states in the skeleton for P_1 with the same atomic proposition valuation: they all assign *false*, *true*, *false* to N_1, T_1, C_1 respectively (these states being shown as circles containing T_1, as per our convention for figures). To distinguish local states with the same atomic proposition valuation, we now number the local states, as discussed in Section 3.2. Figure 14 shows the result of performing this numbering on the program in Figure 13. The numbers are

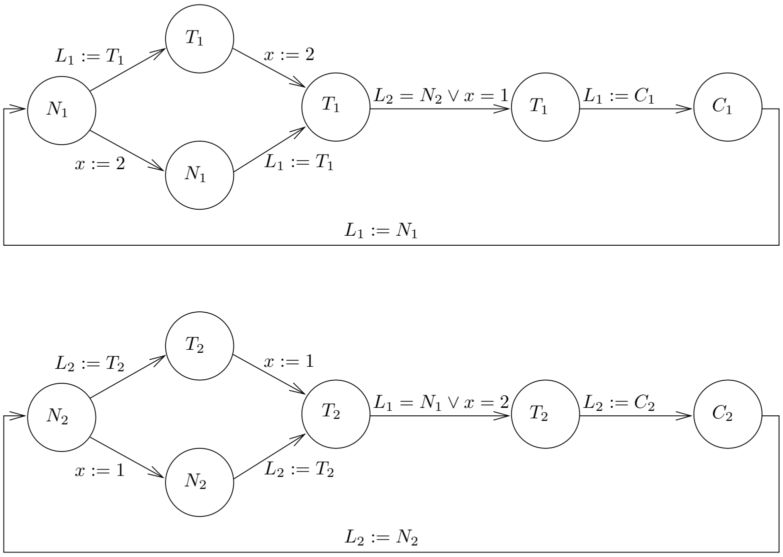


Fig. 13. Decomposed two-process mutual exclusion program.

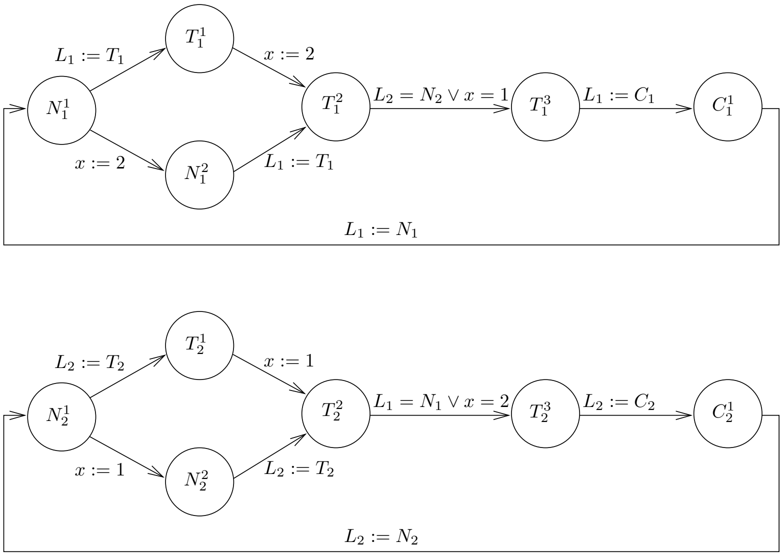


Fig. 14. Decomposed two-process mutual exclusion program after all local states have been numbered.

shown as superscripts on the atomic proposition(s) displayed in each local state. We use this convention of superscripting all the displayed atomic propositions of a local state with the number of that local state.

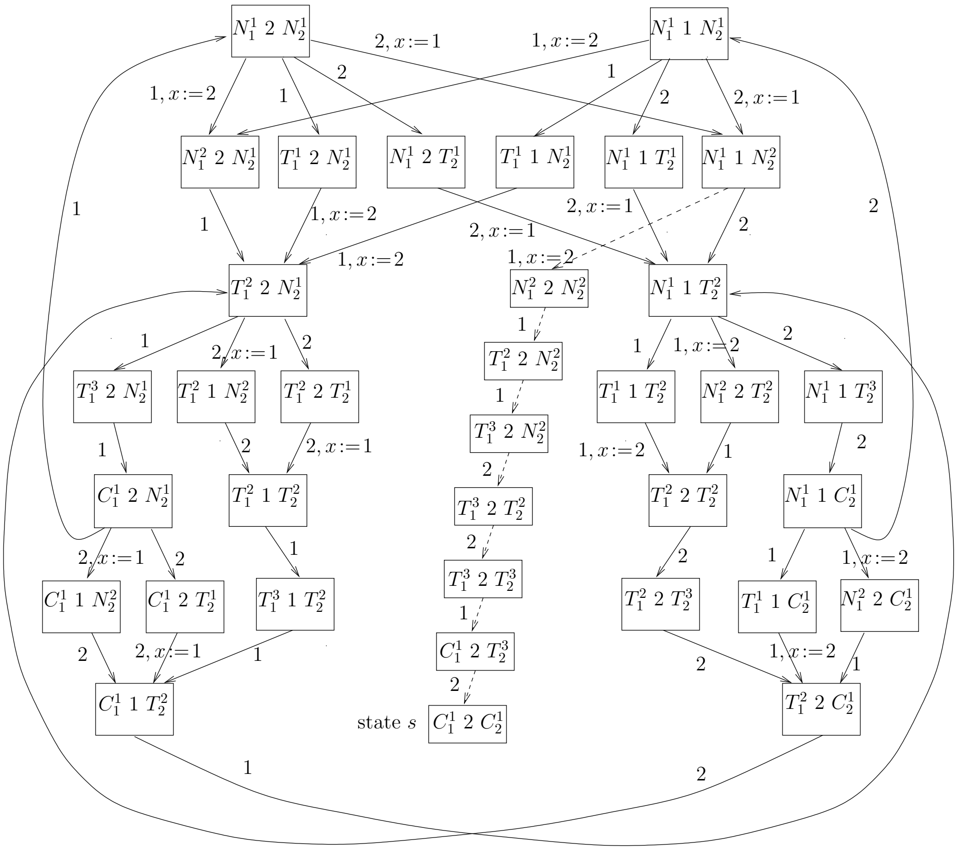


Fig. 15. Partial global-state transition diagram of the program of Figure 14.

5.3 Phase 3

Figure 15 shows (part of) the global-state transition diagram of the program in Figure 14. The solid lines indicate paths that correspond to (decompositions of) paths in Figure 9. These paths represent the same interleavings of transitions as in Figure 9. The single-dashed path shown represents a "new" interleaving, due to the decomposition of the arcs (of the program in Figure 12). There are many more such new paths, but they have been omitted for sake of clarity of the figure. Also, the assignments to L_1, L_2 have been omitted, since they can be easily inferred from the begin and end states of each transition.

5.4 Phase 4

We discuss each step in turn.

Step 4.1: Label the Reachable States. Taking as our specification that given for the two-process mutual exclusion problem in Section 2.4, we see, that in Figure 15, every state will be labeled with $\neg(C_1 \wedge C_2)$. Also, all states satisfying T_1 (T_2) will be labeled with $\mathsf{AF}C_1$ ($\mathsf{AF}C_2$) respectively.

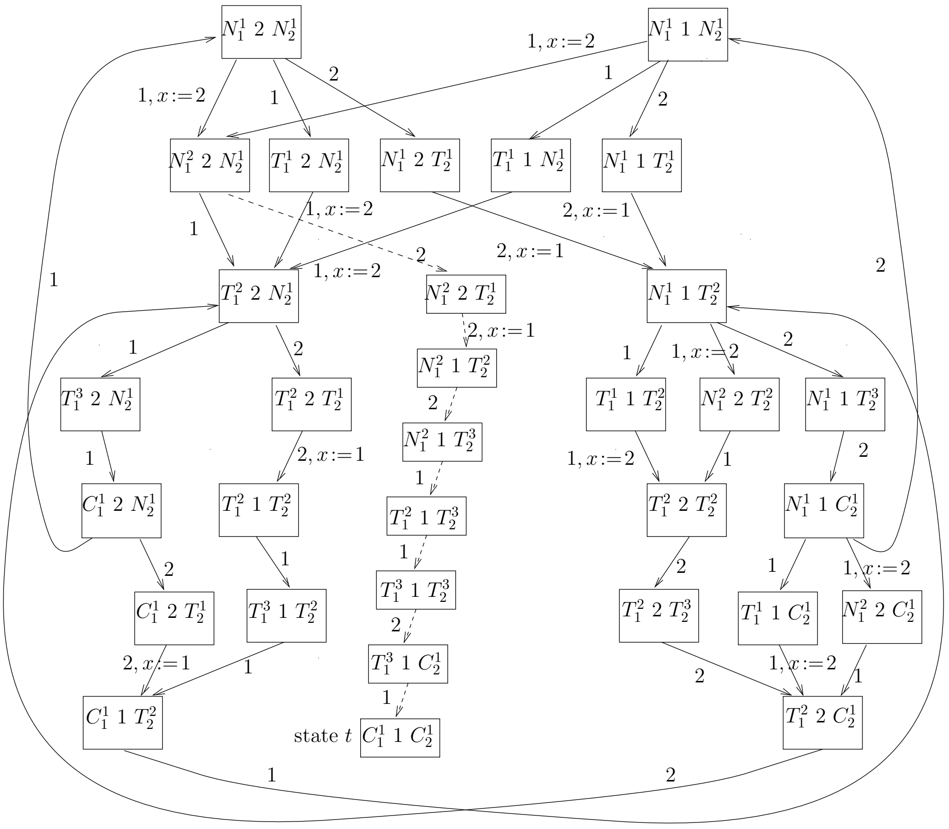


Fig. 16. Partial global-state transition diagram of the program of Figure 14 after some deletions have been performed.

Step 4.2: Prune Structure So that Labels Are Satisfied. Figure 15 contains a state $s = [C_1^1\ 2\ C_2^1]$ (at the end of the dashed path) such that $\neg(C_1 \wedge C_2) \in label(s)$, and $s \not\models \neg(C_1 \wedge C_2)$. Hence, by the **Prop-rule**, s must be made unreachable. A deletable transition along a path from an initial state to s is $[N_1^1\ 1\ N_2^1] \overset{2,x:=1}{\longrightarrow} [N_1^1\ 1\ N_2^2]$. Since this transition is a member of an unguarded family, the entire family must be deleted. Figure 16 shows (part of) the resulting Kripke structure. Again, there is a state t (in Figure 16) such that $\neg(C_1 \wedge C_2) \in label(t)$, and $t \not\models \neg(C_1 \wedge C_2)$, and so t must be made unreachable by the **Prop-rule**. This can be achieved by the deletion of the (unguarded) family containing the transition $[N_1^1\ 2\ N_2^1] \overset{1,x:=2}{\longrightarrow} [N_1^2\ 2\ N_2^1]$. The (reachable part of the) complete Kripke structure that results after all deletions have been made is shown in Figure 17.

5.5 Phase 5

We discuss each step in turn.

Step 5.1: Ensure Atomic Read/Write Form. Figure 17 gives M''' for our ex-

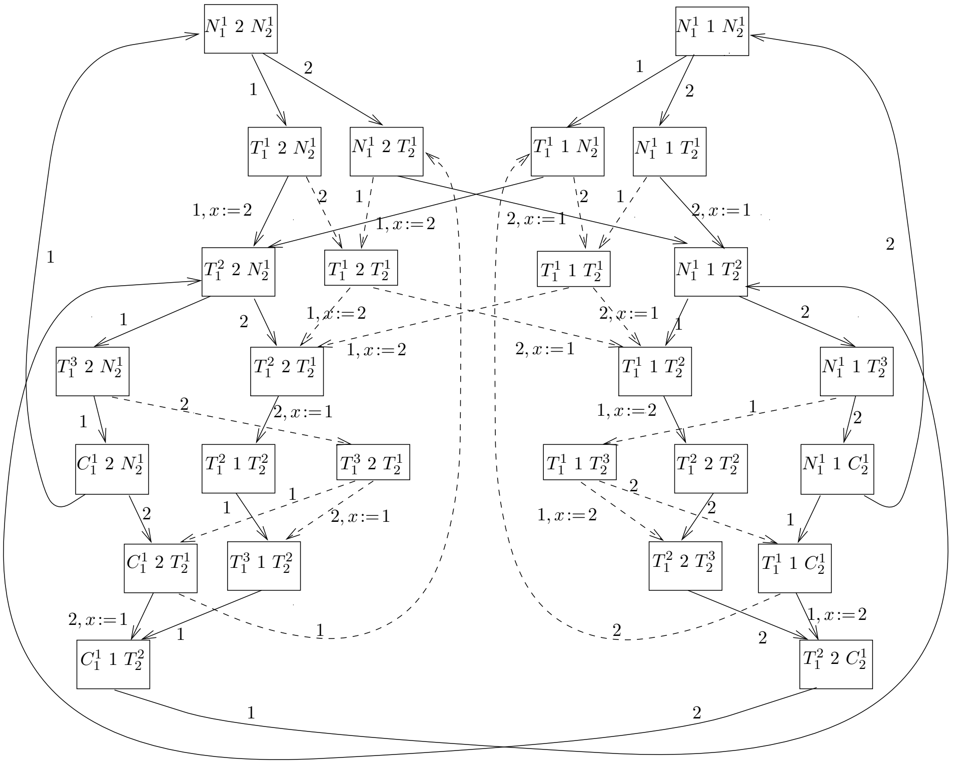


Fig. 17. Global-state transition diagram of the program of Figure 14 after all deletions have been performed.

tended example. There is exactly one guarded and nonwriting P_1-family $\mathcal{F}$ in M''', consisting of the following transitions:

$$[T_1^2 \ 2 \ N_2^1] \xrightarrow{1} [T_1^3 \ 2 \ N_2^1]$$
$$[T_1^2 \ 1 \ T_2^2] \xrightarrow{1} [T_1^3 \ 1 \ T_2^2]$$

We can verify, by inspection, that this family is single-reading. The single P_2-family in M''' that is guarded and nonwriting is also seen to be single-reading. Thus, the function CHECK-STRUCTURE returns *true*, and so no states need be deleted.

Step 5.2: Extract the Final Atomic Read/Write Program. Consider first the guarded and nonwriting P_1-family $\mathcal{F}$ in M''', given above. The only simple terms that do not generate transitions outside $\mathcal{F}$ are "$N_2 \in L_2$" and "$x = 1$." Thus, in the invocation CHECK-FAMILY$(M''', \mathcal{F})$, *ok* is set to {"$N_2 \in L_2$", "$x = 1$"}. Since "$N_2 \in L_2$" generates $[T_1^2 \ 2 \ N_2^1] \xrightarrow{1} [T_1^3 \ 2 \ N_2^1]$, and "$x = 1$" generates $[T_1^2 \ 1 \ T_2^2] \xrightarrow{1} [T_1^3 \ 1 \ T_2^2]$, we see that COV$(\mathcal{F}, ok)$ is true in this case. Hence, CHECK-FAMILY$(M''', \mathcal{F})$ computes the guard "$N_2 \in L_2 \vee x = 1$" for the corresponding extracted arc (this is returned as the final value of $\mathcal{F}.srguard$). Thus the single-reading arc $(T_1^2, N_2 \in L_2 \vee x = 1 \rightarrow skip, T_1^3)$ is extracted.

The unguarded P_1-families in Figure 17 and the corresponding extracted arcs are as follows:

$$\begin{array}{l}
[N_1^1\ 2\ N_2^1] \xrightarrow{1} [T_1^1\ 2\ N_2^1]\\
[N_1^1\ 1\ N_2^1] \xrightarrow{1} [T_1^1\ 1\ N_2^1]\\
[N_1^1\ 2\ T_2^1] \xrightarrow{1} [T_1^1\ 2\ T_2^1]\\
[N_1^1\ 1\ T_2^1] \xrightarrow{1} [T_1^1\ 1\ T_2^1]\\
[N_1^1\ 1\ T_2^2] \xrightarrow{1} [T_1^1\ 1\ T_2^2]\\
[N_1^1\ 1\ T_2^3] \xrightarrow{1} [T_1^1\ 1\ T_2^3]\\
[N_1^1\ 1\ C_2^1] \xrightarrow{1} [T_1^1\ 1\ C_2^1]
\end{array}$$

from which the unguarded arc $(N_1^1, \mathit{true} \to L_1 := \{T_1\}, T_1^1)$ is extracted, and

$$\begin{array}{l}
[T_1^1\ 2\ N_2^1] \overset{1,x:=2}{\longrightarrow} [T_1^2\ 2\ N_2^1]\\
[T_1^1\ 1\ N_2^1] \overset{1,x:=2}{\longrightarrow} [T_1^2\ 2\ N_2^1]\\
[T_1^1\ 2\ T_2^1] \overset{1,x:=2}{\longrightarrow} [T_1^2\ 2\ T_2^1]\\
[T_1^1\ 1\ T_2^1] \overset{1,x:=2}{\longrightarrow} [T_1^2\ 2\ T_2^1]\\
[T_1^1\ 1\ T_2^2] \overset{1,x:=2}{\longrightarrow} [T_1^2\ 2\ T_2^2]\\
[T_1^1\ 1\ T_2^3] \overset{1,x:=2}{\longrightarrow} [T_1^2\ 2\ T_2^3]\\
[T_1^1\ 1\ C_2^1] \overset{1,x:=2}{\longrightarrow} [T_1^2\ 2\ C_2^1]
\end{array}$$

from which the unguarded arc $(T_1^1, \mathit{true} \to x := 2, T_1^2)$ is extracted, and

$$\begin{array}{l}
[T_1^3\ 2\ N_2^1] \xrightarrow{1} [C_1^1\ 2\ N_2^1]\\
[T_1^3\ 2\ T_2^1] \xrightarrow{1} [C_1^1\ 2\ T_2^1]\\
[T_1^3\ 1\ T_2^2] \xrightarrow{1} [C_1^1\ 1\ T_2^2]
\end{array}$$

from which the unguarded arc $(T_1^3, \mathit{true} \to L_1 := \{C_1\}, C_1^1)$ is extracted, and

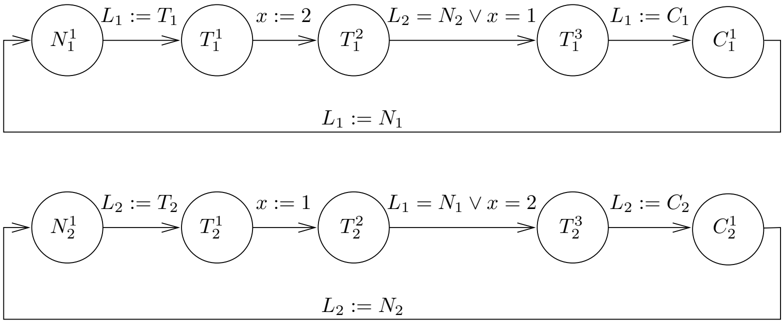
$$\begin{array}{l}
[C_1^1\ 2\ N_2^1] \xrightarrow{1} [N_1^1\ 2\ N_2^1]\\
[C_1^1\ 2\ T_2^1] \xrightarrow{1} [N_1^1\ 2\ T_2^1]\\
[C_1^1\ 1\ T_2^2] \xrightarrow{1} [N_1^1\ 1\ T_2^2]
\end{array}$$

from which the unguarded arc $(C_1^1, \mathit{true} \to L_1 := \{N_1\}, N_1^1)$ is extracted.

These extracted arcs constitute the synchronization skeleton for P_1 in Figure 18. The synchronization skeleton for P_2, also shown in this figure, is extracted from the Kripke structure of Figure 17 in a similar manner. Figure 18 gives the synthesized atomic read/write program for our extended example: the two-process mutual exclusion problem. Note that we use the convention given in the beginning of Section 5.

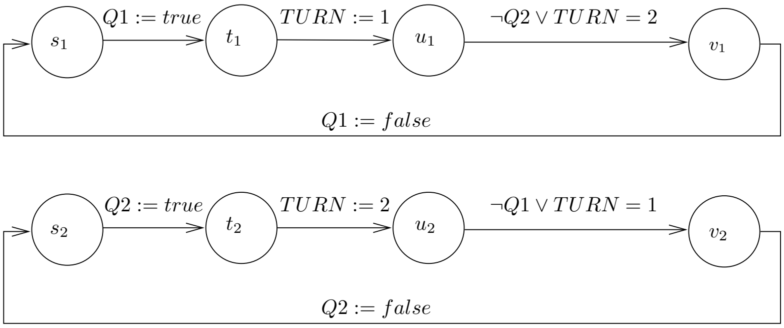
5.6 Comparison with Peterson's Mutual Exclusion Solution

It is instructive to compare our synthesized solution to the two-process mutual exclusion problem, given in Figure 18, with the well-known solution of Peterson [1981], which is shown in Figure 19. s_1, t_1, u_1, v_1 (s_2, t_2, u_2, v_2) are names for local states of P_1 (P_2) respectively. s_1, s_2 are the noncritical states, and v_1, v_2 are the critical states.



The initial state set is { $[L_1 = N_1\ num_1 = 1\ L_2 = N_2\ num_2 = 1\ x = 1]$,
$[L_1 = N_1\ num_1 = 1\ L_2 = N_2\ num_2 = 1\ x = 2]$ }

Fig. 18. Atomic read/write program for the two-process mutual exclusion problem.



The initial state set is { $[s_1\ \neg Q1\ s_2\ \neg Q2\ TURN = 1]$,
$[s_1\ \neg Q1\ s_2\ \neg Q2\ TURN = 2]$ }

Fig. 19. Petersons solution for the two-process mutual exclusion problem.

We see that TURN plays the same role as x, with “flipped” values, so that TURN $= 2$ gives priority to P_1, whereas $x = 1$ gives priority to P_1. We also see that Q1 is a boolean variable that indicates whether or not P_1 is in its neutral state, i.e., $Q1 =$ *false* is “equivalent” to $N_1 =$ *true*. We note that Peterson’s solution only has four local states in each process, whereas ours has five (with one binary-valued shared variable in both solutions). This is explained by noting that our synthesis method is suboptimal in the following respect. Inspecting Figure 11, we note that the only atomic proposition in $\mathcal{AP}_2 = \{N_2, T_2, C_2\}$ that is tested by P_1 is N_2. In other words, N_2 is the only atomic proposition that needs to be “externally visible.” The propositions T_2 and C_2 are not tested by P_1 and so do not have to be externally visible. Thus, we could optimize the synthesis method by “grouping” only the atomic propositions of each process P_i that are referenced by some $P_j, j \neq i$, into the externally visible location counter L_i. The remaining propositions could then be grouped into an internal location counter IL_i. In the mutual exclusion example, we see that L_2 (L_1) would incorporate just N_2 (N_1) respectively, while IL_2 (IL_1) would incorporate T_2 (T_1) and C_2 (C_1) respectively. Thus, for example, the arc $(T_2, N_1 \vee x = 2 \rightarrow \mathit{skip}, C_2)$ in Figure 11 becomes, upon introduction of the location counters, the arc $(T_2, L_1 = N_1 \vee x = 2 \rightarrow IL_2 := C_2, C_2)$, cf. Figure 12. Since IL_2 is not externally visible, a write operation to it does not count as a write to shared data, and so, in the decomposition step of Phase 2, the arc $(T_2, L_1 = N_1 \vee x = 2 \rightarrow IL_2 := C_2, C_2)$ can be left as is, and considered to be a guarded and nonwriting arc, whereas the analogue arc in Figure 12, namely $(T_2, L_1 = N_1 \vee x = 2 \rightarrow L_2 := C_2, C_2)$, has to be decomposed into the guarded and nonwriting arc $(T_2, L_1 = N_1 \vee x = 2 \rightarrow \mathit{skip}, T_2)$, followed by the unguarded and single-writing arc $(T_2, \mathit{true} \rightarrow L_2 := C_2, C_2)$, cf. Figure 13. Avoiding this decomposition would result in a mutual exclusion program with four local states per process, which would be effectively isomorphic to Peterson's program.

Since this optimization is straightforward in principle, and does not add any important capabilities to our method, we omit it for sake of simplicity.

5.7 Another Example: The Barrier Synchronization Problem

In this problem, each process consists of a cyclic sequence of two terminating phases, phase A and phase B. Process i ($i \in \{1,2\}$) is in exactly one of four local states, SA_i, EA_i, SB_i, EB_i, corresponding to the start of phase A, the end of phase A, the start of phase B, and the end of phase B, respectively. The CTL specification is the conjunction of the following:

(1) Initial State (both processes are initially at the start of phase A): $SA_1 \wedge SA_2$

(2) The start of phase A is immediately followed by the end of phase A:
$$\mathsf{AG}(SA_i \Rightarrow \mathsf{AY}_i EA_i)$$

(3) The end of phase A is immediately followed by the start of phase B:
$$\mathsf{AG}(EA_i \Rightarrow \mathsf{AY}_i SB_i)$$

(4) The start of phase B is immediately followed by the end of phase B:
$$\mathsf{AG}(SB_i \Rightarrow \mathsf{AY}_i EB_i)$$

(5) The end of phase B is immediately followed by the start of phase A:
$$\mathsf{AG}(EB_i \Rightarrow \mathsf{AY}_i SA_i)$$

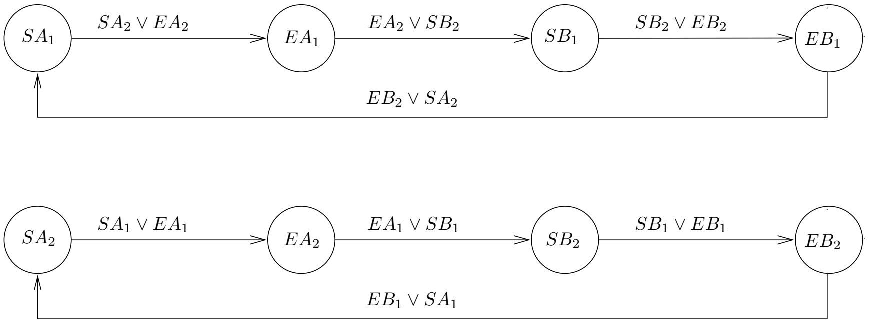
(6) P_i is always in exactly one of the states SA_i, EA_i, SB_i, EB_i:
$$\mathsf{AG}(SA_i \equiv \neg(EA_i \vee SB_i \vee EB_i)) \wedge \mathsf{AG}(EA_i \equiv \neg(SA_i \vee SB_i \vee EB_i)) \wedge$$
$$\mathsf{AG}(SB_i \equiv \neg(SA_i \vee EA_i \vee EB_i)) \wedge \mathsf{AG}(EB_i \equiv \neg(SA_i \vee EA_i \vee SB_i))$$

(7) P_1 and P_2 are never simultaneously at the start of different phases:
$$\mathsf{AG}(\neg(SA_1 \wedge SB_2)) \wedge \mathsf{AG}(\neg(SA_2 \wedge SB_1))$$

(8) P_1 and P_2 are never simultaneously at the end of different phases:
$$\mathsf{AG}(\neg(EA_1 \wedge EB_2)) \wedge \mathsf{AG}(\neg(EA_2 \wedge EB_1))$$

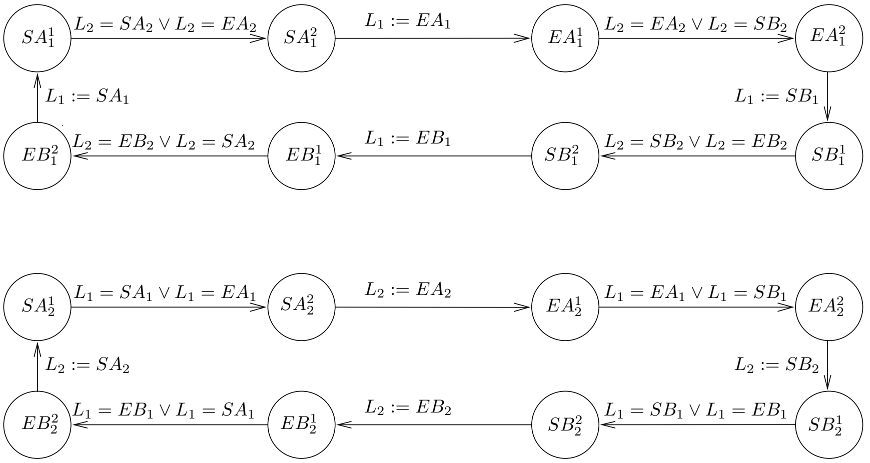
(9) A transition by one process cannot cause a transition by another, $i, j \in \{1,2\}$, $i \neq j$ (interleaving model of concurrency):
$$\mathsf{AG}(SA_i \Rightarrow \mathsf{AY}_j SA_i) \wedge \mathsf{AG}(EA_i \Rightarrow \mathsf{AY}_j EA_i) \wedge$$
$$\mathsf{AG}(SB_i \Rightarrow \mathsf{AY}_j SB_i) \wedge \mathsf{AG}(EB_i \Rightarrow \mathsf{AY}_j EB_i)$$

(10) It is always the case that some process can move: $\mathsf{AGEX}\mathit{true}$



The initial state set is { $[SA_1\ SA_2]$ }

Fig. 20. Program for the barrier synchronization problem.



The initial state set is { $[SA_1^1\ SA_2^1], [SA_1^2\ SA_2^1], [SA_1^1\ SA_2^2], [SA_1^2\ SA_2^2]$ }

Fig. 21. Atomic read/write program for the barrier synchronization problem.

(7) and (8) together specify the synchronization aspect of the problem: P_1 can never get one whole phase ahead of P_2 (and vice-versa). Figure 20 presents the program synthesized by the Emerson and Clarke [1982] method from the barrier synchronization specification. Figure 21 presents the atomic read/write program synthesized from this specification by our synthesis method.

6. TRANSLATING SYNCHRONIZATION SKELETON PROGRAMS INTO PROGRAMS THAT USE ATOMIC REGISTERS

Our current model of computation admits arcs that read or write an externally visible variable (either L_i or a shared variable) and *simultaneously* modify a local variable, namely the variable encoding the local-state numbering function *num*. Hence, strictly speaking, we are not adhering to a *single* atomic read/*single* atomic write model of computation, which allows a *single* atomic read or write operation at a time, whether of a local or a shared variable. Also, the synchronization skeleton model is really based upon the use of an **await** primitive to evaluate guards that reference nonlocal variables: the arc $(s_i, B \rightarrow skip, t_i)$ is essentially the same as an "**await** B" statement. The **await** primitive is generally regarded as a higher-level primitive than atomic reads and writes of registers. This is because implementation of an **await** involves *busy waiting*: the nonlocal variables referenced in the **await** have to be read repeatedly until the **await** condition becomes true (if ever). Thus, as long as the **await** condition is false, the nonlocal variables are being read, but these read operations are "hidden" in that they are not reflected in the transitions generated by the program containing the **await**—the **await** either occurs atomically or not at all. We now present a method of translating the synthesized synchronization skeleton programs into programs that use n-bit atomic registers [Singh et al. 1994] to implement all variables that are accessed by more than one process. We proceed as follows.

Each process P_i''' of P''' is implemented as a single **do** statement D_i (in Dijkstra's guarded command language [Dijkstra 1976]). We make no assumptions about the level of atomicity of the implementation of guarded commands, e.g., the evaluation of guards is not necessarily atomic—it can, for example, be a sequence of operations, which first reads all the referenced variables, and then evaluates the guard. Associated with each D_i are the variables L_i and num_i. L_i is the externally visible location counter, and num_i is an integer variable that contains the number of the current i-state. L_i is readable (but not writable) by $D_j, j \neq i$, and num_i is local to D_i, i.e., not readable or writable by any $D_j, j \neq i$. Additionally, we introduce a (local to D_i) variable LL_i. LL_i keeps track of the externally visible variable L_i, so that we avoid making two references to externally visible variables in a single guarded command (this is necessary for technical reasons). Finally, we also have the shared variables in $\mathcal{SH}$, which are shared by all $D_i, i \in [1 : K]$.

To emphasize the atomic registers model, we use read, write to indicate read and write operations on nonlocal variables (i.e., variables that are visible to more than one D_i, $i \in [1 : K]$, that is, the shared variables in $\mathcal{SH}$ and the $L_i, i \in [1 : K]$). Each $L_i, i \in [1 : K]$, is implemented by a multiple-reader, single-writer, n-bit atomic register, and each $x \in \mathcal{SH}$ is implemented by a multiple-reader, multiple-writer, n-bit atomic register [Singh et al. 1994].

The guarded commands of each D_i are derived as follows.

—An unguarded and single-writing arc in P_i gives rise to a single guarded command. There are two cases, depending on the form of the arc.

(1) If the arc has the form $(s_i, \mathit{true} \rightarrow x := c, t_i)$, then the corresponding guarded command is

$$\begin{aligned} LL_i = \mathcal{L}[s_i] \ \wedge \ num_i = num(s_i) \rightarrow \ & \mathsf{write}(x, c); \\ & num_i := num(t_i). \end{aligned}$$

(2) If the arc has the form $(s_i, \mathit{true} \rightarrow L_i := \mathcal{L}[t_i], t_i)$, then the corresponding guarded command is

$$LL_i = \mathcal{L}[s_i] \ \wedge \ num_i = num(s_i) \rightarrow \mathsf{write}(L_i, \mathcal{L}[t_i]);$$

$$LL_i := \mathcal{L}[t_i];$$
$$num_i := num(t_i).$$

—A single-reading and nonwriting arc $AR_i = (s_i, \bigvee_{k \in [1:n]} b_k \rightarrow skip, t_i)$ in P_i''' gives rise to n guarded commands, one guarded command for each k in $[1 : n]$, as follows. We introduce a local variable ctr_{AR_i} (unique to AR_i), which is a counter modulo n. Its purpose is to ensure that every disjunct of $\bigvee_{k \in [1:n]} b_k$ is tested for truth, so that if $\bigvee_{k \in [1:n]} b_k$ is continuously true, then this is eventually detected. We also introduce a local boolean variable Lb_i (unique to P_i'''), used to temporarily store the "sampled" value of b_k. For each k in $[1 : n]$, we have the following guarded command in D_i

```
$LL_i = \mathcal{L}[s_i] \ \wedge \ num_i = num(s_i) \ \wedge \ ctr = k \rightarrow Lb_i := \mathsf{eval}(b_k);$
                                                    **if** $Lb_i \rightarrow num_i := num(t_i)$
                                                    $[\!]$ $\neg Lb_i \rightarrow ctr := (k \ mod \ n) + 1$
                                                    **fi**.
```

The function eval, whose purpose is to return the value of a simple term in the current global state, is given below. It takes a simple term as argument, reads the shared variable or externally visible location counter in the term, and then returns the value of the term.

```
**boolean function** eval($b$)
/* $b$ is assumed to be either a simple term, or the constant *true*
   $TLL_i$, $Lx_i$ are local variables unique to $P_i'''$ */
**if** $b \doteq$ "$Q_j \in L_j$" $\rightarrow TLL_i :=$ read$(L_j)$;
                          **return**$(Q_j \in TLL_i)$
$[\!]$ $b \doteq$ "$x = c$" $\rightarrow Lx_i :=$ read$(x)$;
                        **return**$(Lx_i = c)$
$[\!]$ $b \doteq$ "*true*" $\rightarrow$ **return**(*true*)
**fi**
```

In practice, eval is most efficiently implemented as an in-line macro. The tests on the syntactic form of b ($b \doteq$ "$Q_j \in L_j$", $b \doteq$ "$x = c$", $b \doteq$ "*true*") can then be dispensed with. It is, however, technically convenient for our purposes to use eval to encapsulate all the read operations.

The translation of $P''' = P_1''' \| \cdots \| P_K'''$ into guarded command notation is then given by $D = D_1 \| \cdots \| D_K$, where each D_i, $i \in [1 : K]$, is the translation of P_i''' given above, and $\|$ still denotes parallel composition under a nondeterministic interleaving model of concurrency. Figure 22 shows the result of applying this translation to the program of Figure 18.

Let $\hat{D} = \hat{D}_1 \| \cdots \| \hat{D}_K$ be a *large-grain* version of D in which the execution of each guarded command is atomic, that is, the computations of $\hat{D}$ are sequences of segments, each segment consisting entirely of all of the operations resulting from a single execution of a single guarded command. Let $M^D = (S_0^D, S^D, R^D)$, $\hat{M}^D = (\hat{S}_0^D, \hat{S}^D, \hat{R}^D)$ be the global-state transition diagrams generated by the execution of $D, \hat{D}$ respectively (the formal definitions of $M^D, \hat{M}^D$ are similar to the global-state transition diagram definition (2.2.2), and are a straightforward exercise in operational semantics). A global state u of D (or $\hat{D}$) is a mapping of the variables

```
$D_1$ :: do
            $LL_1 = N_1 \ \wedge\ num_1 = 1 \ \rightarrow$ write($L_1, T_1$); $LL_1 := T_1$; $num_1 := 1$
         [] $LL_1 = T_1 \ \wedge\ num_1 = 1 \ \rightarrow$ write($x, 2$); $num_1 := 2$
         [] $LL_1 = T_1 \ \wedge\ num_1 = 2 \ \wedge\ ctr_1 = 1 \ \rightarrow$ $Lb_1 :=$ eval($L_2 = N_2$);
                                                        if $Lb_1 \rightarrow num_1 := 3$
                                                        [] $\neg Lb_1 \rightarrow ctr_1 := 2$
                                                        fi
         [] $LL_1 = T_1 \ \wedge\ num_1 = 2 \ \wedge\ ctr_1 = 2 \ \rightarrow$ $Lb_1 :=$ eval($x = 1$);
                                                        if $Lb_1 \rightarrow num_1 := 3$
                                                        [] $\neg Lb_1 \rightarrow ctr_1 := 1$
                                                        fi
         [] $LL_1 = T_1 \ \wedge\ num_1 = 3 \ \rightarrow$ write($L_1, C_1$); $LL_1 := C_1$; $num_1 := 1$; < cs$_1$ >
         [] $LL_1 = C_1 \ \wedge\ num_1 = 1 \ \rightarrow$ write($L_1, N_1$); $LL_1 := N_1$; $num_1 := 1$
         od
||
$D_2$ :: do
            $LL_2 = N_2 \ \wedge\ num_2 = 1 \ \rightarrow$ write($L_2, T_2$); $LL_2 := T_2$; $num_2 := 1$
         [] $LL_2 = T_2 \ \wedge\ num_2 = 1 \ \rightarrow$ write($x, 1$); $num_2 := 2$
         [] $LL_2 = T_2 \ \wedge\ num_2 = 2 \ \wedge\ ctr_2 = 1 \ \rightarrow$ $Lb_2 :=$ eval($L_1 = N_1$);
                                                        if $Lb_2 \rightarrow num_2 := 3$
                                                        [] $\neg Lb_2 \rightarrow ctr_2 := 2$
                                                        fi
         [] $LL_2 = T_2 \ \wedge\ num_2 = 2 \ \wedge\ ctr_2 = 2 \ \rightarrow$ $Lb_2 :=$ eval($x = 2$);
                                                        if $Lb_2 \rightarrow num_2 := 3$
                                                        [] $\neg Lb_2 \rightarrow ctr_2 := 1$
                                                        fi
         [] $LL_2 = T_2 \ \wedge\ num_2 = 3 \ \rightarrow$ write($L_2, C_2$); $LL_2 := C_2$; $num_2 := 1$; < cs$_2$ >
         [] $LL_2 = C_2 \ \wedge\ num_2 = 1 \ \rightarrow$ write($L_2, N_2$); $LL_2 := N_2$; $num_2 := 1$
         od
```

Fig. 22. Atomic register guarded commands program for two-process mutual exclusion. < cs$_1$ >, < cs$_2$ > are the critical sections of D_1, D_2, respectively.

of D onto their appropriate domains. S^D is the set of all such global states. $\hat{S}^D$ is the set of all such global states that assign the same values to L_i and LL_i, for all $i \in [1 : K]$.

Define an *empty transition of* $\hat{M}^D$ to be a transition of $\hat{M}^D$ in which the eval function is invoked, and returns *false*. Thus an empty transition tests a disjunct of some guard in some process P_i''' and finds it false, and so no P_i'''-transition is "simulated" by $\hat{D}_i$. It is possible, therefore, to have fullpaths with a suffix consisting entirely of (the interleaved operations of) empty transitions, which means that no real progress in the computation of P''' is being made. The only thing that is occurring is that the same false guards are being tested over and over again.[19] Likewise, define an *empty transition of* M^D to be a transition of M^D resulting from an execution of the eval function which returns *false* (recall that in M^D there will be several transitions resulting from a single execution of eval, whereas in $\hat{M}^D$, execution of eval induces exactly one transition, i.e, there is a one-to-one correspondence).

[19] If $\bigvee_{k \in [1:n]} b_k$ is false, then all of the b_k, $k \in [1 : n]$ are false. Hence eval(b_k) returns false for all $k \in [1 : n]$.

We therefore impose the following *progress condition* on the interleaving of operations of the D_i (in both $\hat{M}^D$ and M^D).

Progress Condition. *No fullpath of $\hat{M}^D, M^D$ has a suffix composed entirely of empty transitions.*

The progress condition is a restriction only on the infinite behavior of $\hat{M}^D, M^D$—it cannot be violated by any finite prefix of some path in $\hat{M}^D, M^D$. In form, therefore, it resembles a *fairness notion* [Francez 1986]. Its sole purpose is to rule out infinite paths in which (one or more) false guards in P''' are tested continuously from some point onward.

The following theorem presents our main correctness result for the atomic-registers-based implementation. It essentially states that the guarded command implementation D satisfies the original problem specification (with the local structure specifications appropriately modified).

Theorem 6.1 (Soundness Theorem for Registers Model). *If $S_0^D \neq \emptyset$, and f is a conjunct of the given specification, and the progress condition holds, then D is an atomic registers program that satisfies f^*, where $f^* = f$ if f has one of the forms h, $\mathsf{AG}h$, $\mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$, and $f^* = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i(p_i \vee q_i))$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$ if $f = \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$ respectively, and where $p, q, r, h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$, $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$.*

The proof is omitted due to lack of space, and can be found in [Attie 1995]. Essentially, we show that M^{iv} and $\hat{M}^D$ agree on all CTL formulae of the form f^*, and so do $\hat{M}^D$ and M^D. The relation between M^{iv} and $\hat{M}^D$ is established by showing that there is a one-to-one correspondence between a transition in M^{iv} that arises from the execution of some arc AR_i of P''', and the transition in $\hat{M}^D$ that arises from the execution of the guarded command in D that is derived from AR_i. This establishes a bisimulation between M^{iv} and $\hat{M}^D$, from which it follows that M^{iv} and $\hat{M}^D$ satisfy the same CTL formulae [Browne et al. 1988]. The relation between $\hat{M}^D$ and M^D is established by using the results of Lamport [1990], which shows that under certain conditions a class of correctness properties is preserved when a sequence of actions is combined into a single atomic action (provided that the sequence contains only a single access to a single shared variable), or the atomic action is decomposed into the sequence. The main result of Lamport [1990] then implies that $\hat{M}^D$ and M^D agree on all formulae in our specfication language. The reader is referred to Attie [1995] for details of the proof.

Applying Proposition 4.5.2.1 then allows us to conclude that D satisfies f^*. Furthermore, D is an atomic registers program by construction of the translation procedure from synchronization skeleton notation given above.

6.1 Implementing the Progress Condition

We now show that implementing the progress condition does not unduly restrict the scheduling of D.

Claim. Let u be an arbitrary reachable state in M^D, and let π^D be an arbitrary maximal path of M^D starting in u. Then there exists a state v along π^D, and a guarded command GC in D, such that $v(GC.guard) = true$, and execution of GC in v results in a nonempty transition.

A proof of this claim is provided in Attie [1995].

Thus, we see that implementing the progress condition does not restrict the scheduling of D (except to prohibit infinite empty paths), since, from any reachable state, no matter which path is chosen by the scheduler, it will always be possible to satisfy the progress condition, i.e., to eventually execute a nonempty transition.

In practice, the progress condition could be implemented by adding to each D_i a variable $flag\text{–}empty_i$, which would be set to *true* whenever eval returns *false*, and set to *false* whenever eval returns *true* or a write is executed. The scheduler would then check for states in which $\bigwedge_{i\in[1:K]} flag\text{–}empty_i$ is true. All that is required is to attempt to detect $\bigwedge_{i\in[1:K]} flag\text{–}empty_i$ under the assumption that it is eventually stable. In the event of detection, the scheduler must intervene, but may wait arbitrarily long to do so. Intervention takes the form of finding an enabled guarded command whose execution leads to a nonempty transition, (from the claim, there must be at least one such guarded command) and executing it.

7. ANALYSIS AND DISCUSSION

7.1 Space Complexity

Consider the finite-state model which is input to Phase 1. M is either given directly, as input to our synthesis method, or is generated by the synthesis method of Emerson and Clarke [1982], from a CTL specification. In the case that M is generated by the Emerson and Clarke [1982] method, a shared variable is introduced whenever two or more global states occur that are propositionally equivalent, but not temporally equivalent in the sense that they differ in some of the temporal formulae that they satisfy (see Section 2.6). The shared variable serves to distinguish such states from each other. We shall assume that a directly given M also has this characteristic. This is a reasonable assumption, since states that are propositionally inequivalent can always be distinguished by reading the externally visible location counters (which encode the values of the atomic propositions), and so no shared variable is needed to distinguish these states from each other. In the worst case, the number of sets of propositionally equivalent but temporally inequivalent states is linear in the size of M, and therefore so is the number of shared variables. Hence, the refinement of the model M effected by Phases 1 and 2 can lead to an exponential blowup, causing the method to have (at least) double-exponential space complexity. To avoid this, we note that any finite-state model M can be easily modified so that it uses at most one shared variable.

Let $\bar{s}, \bar{t}$ denote different sets of propositionally equivalent but temporally inequivalent global states in M. Let x_s, x_t denote the shared variables that are introduced to distinguish all the occurrences of global states in $\bar{s}, \bar{t}$ respectively. x_s is written only upon entry to states in $\bar{s}$ and read only upon exit from states in $\bar{s}$, likewise for x_t. Since $\bar{s}$ and $\bar{t}$ have no common members (by definition), it should be possible to "reuse" x_s to distinguish among the global states in $\bar{t}$. Even in the case where there is a transition from a state in $\bar{s}$ to a state in $\bar{t}$ this would work: the transition would be labeled with a test (to determine which state in $\bar{s}$ is in fact the current global state) and set (to record which global state in $\bar{t}$ is being entered). We refer the reader to Emerson and Clarke [1982] for more discussion of this technique, while also noting that it is applicable to directly given models M, as well as those generated by the Emerson and Clarke [1982] synthesis method.

With only one shared variable present, we see that once all atomic propositions of a process have been consolidated into the externally visible location counter, then the largest multiple-assignments that could occur would have size two. Hence, in the decomposition carried out in Phase 2, each local state can be refined into at most five local states (with the same propositional valuation but different local state numbers—see Section 3.2). This is because one new local state is introduced for the "test," and three new local states are introduced for the two serializations of the multiple-assignment.

Suppose that the high-atomicity program extracted from M in Phase 1 has K processes with $O(N)$ local states each. By Definition 2.3, every local state in (some process P_i of) the program is the projection onto P_i of some global state in M. Thus, M contains $O(N^K)$ states. By the previous paragraph, in any intermediate program generated by the method, (including the final atomic read/write program), there are K processes each containing $O(5N)$ local states, i.e., $O(N)$ local states. Thus, the largest intermediate global-state transition diagram generated by our method contains $O(N^K)$ states. Thus, our method has space complexity $O(N^K)$, i.e., the same order of (space) complexity as the finite-state model M that is input to the method. This is clearly the best that we can do.

7.2 Related Work

The refinement of concurrent programs is a topic of extensive research, with many proposed approaches. The most general approach constructs a low-atomicity program directly, and then establishes that the low-atomicity program is a "correct implementation" of the high-atomicity program. Different approaches are distinguished by their notion of "correct implementation." For example, in the I/O automaton approach [Lynch and Tuttle 1987; Lynch and Vaandrager 1995], a low-atomicity program is a correct implementation if each of its externally observable behaviors (traces) is also a trace of the high-atomicity program. The approach of Back and von Wright [1994] for refinement of action systems is likewise based on trace-inclusion. The proofs of correct implementation are based on exhibiting a simulation relation between the low-atomicity and high-atomicity programs. Simulation is also used to establish refinement in the refinement calculus [Back 1989]. In UNITY [Chandy and Misra 1988], refinement is performed on specifications, expressed as UNITY formulae, and a refined specification is correct if it logically implies the higher-level specification.

In some approaches, the low-atomicity program is shown to have some properties that are derived from properties established for the high-atomicity program. (Our method uses a similar idea for the local-structure properties $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$ and $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$—see Theorems 4.5.2.2 and 6.1.) Manna and Pnueli [1995, Chapter 1] shows how an invariant for the low-atomicity program can be derived as a refinement itself of an invariant for the high-level program. Other methods for refining a high-atomicity program together with its invariant are given in Gribomont [1990], Gribomont [1993]. Here, each time the program is refined, the invariant that holds for the refined program is derived systematically from the original program, the refined program, and the invariant that holds for the original program.

While some of the above methods provide a methodology for deriving the low-atomicity program from the high-atomicty one, there is little guidance as to which choices of refinement step will provide a correct refinement. This usually has to be established afterward, e.g., by attempting to construct a simulation relation, or a proof of invariance. Our method provides an initial "naive" refinement, in which all possible refinements of each process action are performed initially. Then, our method provides guidance in eliminating refinements that cause the specification to be violated (see the deletion rules, Figure 7).

The only other temporal logic synthesis method to date that we are aware of which generates atomic read/write programs is that of Dill and Wong-Toi [1990] for synthesizing reactive modules. This method has double-exponential space complexity in the length of the specification, and produces a program that has size double-exponential in the length of the specification. By comparison, our method has single-exponential space complexity in the length of the specification. Furthermore, while the programs we synthesize could in the worst case have length exponential in the specification, this would occur only if one process was much larger than the others, since the size of the (exponentially large) model is on the order of the product of the sizes of the processes. Thus, in practice, we expect our programs to have size polynomial in the length of the specification.

On the other hand, the method of Dill and Wong-Toi [1990] is complete whereas ours is not. While interesting theoretically, this advantage of Dill and Wong-Toi [1990] is unlikely to have practical significance because the double-exponential space complexity is a considerable impediment to practical application.

7.3 Further Directions for Research

Our method may easily be extended to handle temporal modalities other than those given in Section 2.4. All that is needed is to devise the appropriate labeling and deletion rules for Phase 4. All the other phases of the method are independent of the particular subset of CTL chosen as the specification language. Furthermore, it may be possible to extend our method to synthesize shared-memory concurrent programs for different synchronization primitives such as compare-and-swap, fetch-and-add, or n-register assignment [Herlihy 1991]. The main step would be devising a characterization of the Kripke structures corresponding to programs containing the particular primitive, much like our atomic read/write Kripke structures correspond to atomic read/write programs. The refinement phases (1 and 2) would then have to be modified to refine the initial model into a structure that (after the deletion rules have been applied) is in the appropriate form.

One potential drawback of our method is its susceptibility to *state-explosion.* Our method requires generation of the global-state transition diagram. For a program consisting of K processes each with $O(N)$ local states, the size of the global-state transition diagram is $O(N^K)$. State-explosion can be dealt with by integrating our work here with that of Attie and Emerson [1998], Attie [1999]. In that method, construction of the exponentially large global-state transition diagram is avoided. Instead, it constructs a state transition diagram for each pair of component processes (of the program) that interact (call these neighbors). This reduces the space complexity to $O(K^2N^2)$. A "pair-program," which embodies all the interactions of the two processes, is then extracted from this state transition diagram. These pair-programs are then composed in a certain way to generate the final program. Integrating the two methods will result in a method that can synthesize large atomic read/write programs efficiently. There are some nontrivial issues however. For example, the composition of the pair-programs relies on a process being able to interact in one atomic step with all of its neighbors. This high atomicity will have to be refined in some way, in the combined method.

Another possible extension of our work would deal with synthesizing fault-tolerant programs. We have already devised a method [Arora et al. 1998] for synthesizing fault-tolerant programs using high-atomiciy operations. Integrating this with our work here would allow the synthesis of fault-tolerant atomic read/write programs, or more generally, fault-tolerant programs that use synchronization primitives such as compare-and-swap. In particular, by specifying that all but one process can crash, we could deal with the important case of synthesizing wait-free programs.

7.4 Conclusions

We have presented a method for the synthesis of atomic read/write programs from specifications expressed in temporal logic. The method is sound but not complete. Although automatic in principle, some of the steps involved require a large amount of search. For example, in the deletion step of Phase 4, there are in general many choices of transitions to be deleted. Thus the method may be best implemented as an interactive tool, akin to a theorem prover, allowing human guidance in order to cut down on the search. Designing good heuristics for selecting transitions for deletion is a topic for future work.

A shortcoming of our method is its incompleteness. This incompleteness arises because our method starts from a coarse high-level program that embodies a particular solution to the specified problem. While this particular solution may be realizable using high-atomicity primitives, it may not be realizable using atomic reads and writes. On the other hand, there may be other solutions to the problem (essentially different from the particular high-level program that our method starts with) which may be realizable using atomic reads and writes. Another way of saying this is that our method embodies one particular strategy for refining a progam from high-level atomicity to atomic read/write atomicity. Other strategies are possible, and these might generate solutions that our method would miss.

A complete method would have a wider scope of applicability. Extending the method to make it complete is thus of some interest. On the other hand, completeness is not essential to practical applicability, as our synthesis of Peterson's solution shows.

Recall that given a specification consisting of a finite-state machine M and CTL formula f, it is not necessary that $M, S_0 \models f$ (S_0 is the set of initial states of M), since the method deletes any states/transitions that cause a violation of f. In the "normal" refinement scenario where $M, S_0 \models f$, states/transitions that cause a violation of f can only be introduced by the decomposition step (Phase 2). However, if $M, S_0 \not\models f$, then such states/transitions could be present initially. In this case, we can regard the synthesis method as providing both refinement and *debugging*.

The work presented in this paper is one building block in our research program, which aims to create truly practical synthesis methods. We have now devised methods that each deal with one aspect of realistic concurrent programs: a realistic, low-atomicity model of concurrent computation (this paper), state-explosion [Attie and Emerson 1998; Attie 1999], and fault-tolerance [Arora et al. 1998]. Some other topics to be dealt with are real-time, and synthesis of asynchronous message-passing distributed programs. The ultimate goal is to create a toolkit of synthesis methods that address many issues in the derivation of large, complex, distributed systems from formal specifications.

APPENDIX

A. PROOFS

Proof of Lemma 3.4.3. The proof is by double implication.

$\underline{s \xrightarrow{i,A} t \in R^1 \text{ follows from } s \xrightarrow{i,A} t \in R^2}$. Let $s \xrightarrow{i,A} t$ be an arbitrary P_i-transition in R^2. Hence, by the global-state transition diagram definition (2.2.2), there is some arc $AR_i = (s{\uparrow}i, B \rightarrow A, t{\uparrow}i)$ in P_i such that $s(B) = \mathit{true}$. By Proposition 3.4.2 and the atomic read/write program definition (3.1.4), AR_i is either single-reading and nonwriting, or unguarded and single-writing. Thus there are two cases.

Case 1: *AR_i is unguarded and single-writing.* By Definition 3.4.1, there must be an unguarded and single-writing P_i-family $\mathcal{F}$ in M such that $\mathcal{F}.\mathit{start} = s{\uparrow}i$, $\mathcal{F}.\mathit{finish} = t{\uparrow}i$, $\mathcal{F}.\mathit{assig} = A$. So, by the unguarded family definition (3.3.2), $s \xrightarrow{i,A} u \in R^1$, for some state u such that $u{\uparrow}i = \mathcal{F}.\mathit{finish}$. (End of case 1.)

Case 2: *AR_i is guarded and nonwriting.* By Definition 3.4.1, there must be a guarded and nonwriting P_i-family $\mathcal{F}$ in M such that $\mathcal{F}.\mathit{start} = s{\uparrow}i$ and $\mathcal{F}.\mathit{finish} = t{\uparrow}i$. Also B, the guard of AR_i, is, by Definition 3.4.1 and Definition 3.3.3, $\bigvee_{k \in [1:n]} b_k$, where $\mathcal{F} = \bigcup_{k \in [1:n]} \mathcal{F}_k$, and conditions (G1), (G2) of Definition 3.3.3 are satisfied. We showed above that $s(B) = \mathit{true}$. Hence $s(b_\ell) = \mathit{true}$ for some $\ell \in [1:n]$. Since, by assumption, s is reachable in M, we can instantiate (G2) for state s, and then take the contrapositive, which yields

$$\text{if } s(b_\ell) = \mathit{true} \text{ then } s \text{ is the start state of some transition in } \mathcal{F}_\ell.$$

Since $\mathcal{F}_\ell \subseteq \mathcal{F}$, we have $s \xrightarrow{i,A} u \in R^1$ for some state u such that $u{\uparrow}i = \mathcal{F}.\mathit{finish}$. (End of case 2.)

Thus in both cases, $s \xrightarrow{i,A} u \in R^1$ for some state u such that $u{\uparrow}i = \mathcal{F}.\mathit{finish}$. By the assumption on Kripke structures in Section 2.3, we have

$$\langle s{\uparrow}\mathcal{SH}\rangle\, A\, \langle u{\uparrow}\mathcal{SH}\rangle \text{ and } \forall j \in [1:K] - \{i\}, s{\uparrow}j = u{\uparrow}j. \qquad \text{(a)}$$

Also in both cases, $s \xrightarrow{i,A} t \in R^2$ (by assumption) and $t{\uparrow}i = \mathcal{F}.\mathit{finish}$. By the global-state transition diagram definition (2.2.2), we have

$$\langle s{\uparrow}\mathcal{SH}\rangle\, A\, \langle t{\uparrow}\mathcal{SH}\rangle \text{ and } \forall j \in [1:K] - \{i\}, s{\uparrow}j = t{\uparrow}j. \qquad \text{(b)}$$

Since A is either *skip* or a deterministic multiple assignment, we have, from the

Hoare triples in (a) and (b), $u{\uparrow}\mathcal{SH} = t{\uparrow}\mathcal{SH}$. We also have, again from (a) and (b), $\forall j \in [1 : K] - \{i\}, u{\uparrow}j = t{\uparrow}j$. Finally, $u{\uparrow}i = \mathcal{F}.\mathit{finish} = t{\uparrow}i$. Hence, by definition of a global state, we conclude $u = t$. Since $s \xrightarrow{i,A} u \in R^1$, we conclude $s \xrightarrow{i,A} t \in R^1$.

$\underline{s \xrightarrow{i,A} t \in R^1 \text{ implies } s \xrightarrow{i,A} t \in R^2}$. Let $T_i = s \xrightarrow{i,A} t$ be an arbitrary P_i-transition in R^1. Hence, by the P_i-family definition (2.2), there is some P_i-family $\mathcal{F}$ in M such that $T_i \in \mathcal{F}$. By Proposition 4.4.2.2, $\mathcal{F}$ is either guarded and nonwriting or unguarded and single-writing. Thus there are two cases.

Case 1: *$\mathcal{F}$ is unguarded and single-writing.* By Definition 3.4.1, there must be a single-writing arc $AR_i = (s{\uparrow}i, \mathit{true} \to A, t{\uparrow}i)$ in P. Thus, by the global-state transition diagram definition (2.2.2), $s \xrightarrow{i,A} u \in R^2$ for some state u such that $u{\uparrow}i = t{\uparrow}i$. (End of case 1.)

Case 2: *$\mathcal{F}$ is guarded and nonwriting.* By Definition 3.4.1, there must be a nonwriting arc $AR_i = (s{\uparrow}i, B \to \mathit{skip}, t{\uparrow}i)$ in P. Now B, the guard of AR_i, is, by Definition 3.4.1 and Definition 3.3.3, $\bigvee_{k\in[1:n]} b_k$, where $\mathcal{F} = \bigcup_{k\in[1:n]} \mathcal{F}_k$, and conditions (G1), (G2) of Definition 3.3.3 are satisfied. Now since $T_i \in \mathcal{F}$, we have $T_i \in \mathcal{F}_\ell$ for some $\ell \in [1 : n]$. Since, by assumption, s is reachable in M, we can instantiate (G1) for T_i to obtain (since $s = T.\mathit{begin}$)

$$s(b_\ell) = \mathit{true}.$$

Therefore, $s(B) = \mathit{true}$, since b_ℓ is a disjunct of B. Hence, by the global-state transition diagram definition (2.2.2), $s \xrightarrow{i,A} u \in R^2$, for some state u such that $u{\uparrow}i = t{\uparrow}i$. (End of case 2.)

Thus in both cases, $s \xrightarrow{i,A} u \in R^2$ for some state u such that $u{\uparrow}i = t{\uparrow}i$. By the global-state transition diagram definition (2.2.2), we have

$$\langle s{\uparrow}\mathcal{SH}\rangle\, A\, \langle u{\uparrow}\mathcal{SH}\rangle \text{ and } \forall j \in [1 : K] - \{i\}, s{\uparrow}j = u{\uparrow}j. \tag{c}$$

Also in both cases, $s \xrightarrow{i,A} t \in R^1$ by assumption. By the assumption on Kripke structures in Section 2.3, we have

$$\langle s{\uparrow}\mathcal{SH}\rangle\, A\, \langle t{\uparrow}\mathcal{SH}\rangle \text{ and } \forall j \in [1 : K] - \{i\}, s{\uparrow}j = t{\uparrow}j. \tag{d}$$

Since A is either *skip* or a deterministic multiple assignment, we have, from the Hoare triples in (c) and (d), $u{\uparrow}\mathcal{SH} = t{\uparrow}\mathcal{SH}$. We also have, again from (c) and (d), $\forall j \in [1 : K] - \{i\}, u{\uparrow}j = t{\uparrow}j$. Hence, by $u{\uparrow}i = t{\uparrow}i$ and the definition of a global state, we conclude $u = t$. Since $s \xrightarrow{i,A} u \in R^2$, we conclude $s \xrightarrow{i,A} t \in R^2$. □

Proof of Proposition 3.4.4. First, we establish P1, P2 and P3:

$$\begin{aligned}&\text{If } \pi \text{ is a finite initialized path in } M^1, \text{ then}\\ &\quad \pi \text{ is also a finite initialized path in } M^2.\end{aligned} \tag{P1}$$

Proof of P1. The proof is by induction on $|\pi|$, the length of π. For the base case, $|\pi| = 0$, and so $\pi = s^0$, where s^0 is some state in S_0. Since s^0 is reachable in $M^2 = (S_0, S, R^2)$ by definition, π is a finite initialized path in M^2 in this case. For the induction step, we assume the induction hypothesis for π a finite path from some state s^0 in S_0 to some state t, and establish P1 for π' consisting of π extended with an arbitrary transition $t \xrightarrow{i,A} t' \in R^1$. Since t lies on a finite initialized path in M^2 (by the induction hypothesis), we have that t is reachable in both M^1, M^2. Hence we can apply Lemma 3.4.3 to conclude $t \xrightarrow{i,A} t' \in R^2$. Since π is a finite initialized path in M^2 (by the induction hypothesis), and $t \xrightarrow{i,A} t' \in R^2$, we have that π' is a finite initialized path in M^2, which concludes the induction step. □

$$\begin{array}{l}\text{If } \pi \text{ is a finite initialized path in } M^2\text{, then}\\ \quad \pi \text{ is also a finite initialized path in } M^1.\end{array} \tag{P2}$$

Proof of P2. The proof is exactly symmetric to the proof of P1 above, with the roles of M^1 and M^2 interchanged. □

$$s \text{ is a reachable state of } M^1 \text{ iff } s \text{ is a reachable state of } M^2. \tag{P3}$$

Proof of P3. The proof is by double implication. For the left-to-right direction, let s be an arbitrary reachable state of M^1. Thus, by definition, there is a finite initialized path π in M^1 which ends in s. By P1 above, π is a finite initialized path in M^2. Thus, by definition, s is a reachable state of M^2. The right-to-left direction is proven in the same way, with the roles of M^1 and M^2 interchanged, and P2 being invoked instead of P1. □

From P3, we see that M^1 and M^2 have the same reachable states. Thus, from Lemma 3.4.3, M^1 and M^2 have the same reachable transitions. Since, by CTL semantics, $M^1, S_0 \models f$ and $M^2, S_0 \models f$ depend only on the reachable portions of M^1 and M^2 respectively, the proposition follows. □

Proof of Proposition 4.4.2.1. Let f be an arbitrary conjunct of the specification. By Section 2.4, f has one of the forms given in the statement of the proposition. Thus there are five cases:

Case 1: *f is of the form h, where $h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$.* Let s be an arbitrary state of S_0''''. By construction of Phase 4, $S_0'''' \subseteq S_0'$. Hence $s \in S_0'$. Thus, by the definition of S_0' (Section 4.1.2), there is some state $s' \in S_0$ such that s is propositionally equivalent to s'. By the soundness of the Emerson and Clarke [1982] synthesis method, $M, S_0 \models h$, and so $s' \models h$. (If $M = (S_0, S, R)$ is given directly, then we assume that all initial states of M satisfy h. In other words, all initial states that violate the initial state specification h are a priori removed from S_0. In particular, if h is derived from M as shown in Figure 4, then all initial states of M will satisfy h by construction.) Hence $s \models h$, since h is purely propositional. Since s is an arbitrarily chosen state of S_0''', we have $M''', S_0''' \models h$ by CTL semantics.

Case 2: *f is of the form $\mathsf{AG}h$, where $h \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$.* Let s be an arbitrary reachable state of M'' such that $s \not\models h$. Since $\mathsf{AG}h$ is a conjunct of the specification, h will be added to $label(s)$ in Phase 4 (by construction of Phase 4—see Figure 5).

Hence s will be deleted or made unreachable in Phase 4 (by construction of Phase 4, in particular, the **Prop-rule**—see Figure 7). Thus, every reachable state of M''' satisfies h. Hence $M''', S_0''' \models \mathsf{AG}h$ by CTL semantics.

Case 3: *f is of the form $\mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$, where $p, q, r \in \mathcal{LO}(\mathcal{AP}, \neg, \wedge)$.* Let s be an arbitrary reachable state of M'' such that $s \models p \wedge \neg\mathsf{A}[q\mathsf{U}r]$. Since $\mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$ is a conjunct of the specification, $\mathsf{A}[q\mathsf{U}r]$ will be added to $label(s)$ in Phase 4 (by construction of Phase 4—see Figure 5). Hence, in Phase 4, by application of the AU**-rule** (Figure 7), every fullpath π starting in s such that $\pi \not\models [q\mathsf{U}r]$ will have one of its transitions deleted, and so M''' will not contain fullpath π. When Phase 4 has terminated, every fullpath starting in s (if any) will satisfy $[q\mathsf{U}r]$. If there are no fullpaths starting in s, then s will be made unreachable in M''' (or will be deleted) by repeated applications of the **EX-rule** (Figure 7), since every path from s terminates in a state with no successors. Hence we conclude that every reachable state in M''' which satisfies p also satisfies $\mathsf{A}[q\mathsf{U}r]$. Thus $M''', S_0''' \models \mathsf{AG}(p \Rightarrow \mathsf{A}[q\mathsf{U}r])$ by CTL semantics.

Case 4: *f is of the form $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, where $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$.* Let s be an arbitrary reachable state of M'' such that $s \models p_i \wedge \neg\mathsf{EX}_i(p_i \vee q_i)$. Since $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$ is a conjunct of the specification, $\mathsf{EX}_i(p_i \vee q_i)$ will be added to $label(s)$ in Phase 4 (by construction of Phase 4—see Figure 5). Hence, by application of the $\mathbf{EX}_i$**-rule** (Figure 7), s will be deleted or made unreachable in Phase 4. Hence we conclude that every reachable state in M''' which satisfies p_i also satisfies $\mathsf{EX}_i(p_i \vee q_i)$. Thus $M''', S_0''' \models \mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i(p_i \vee q_i))$ by CTL semantics.

Case 5: *f is of the form $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$, where $p_i, q_i \in \mathcal{LO}(\mathcal{AP}_i, \neg, \wedge)$.* By the soundness of the Emerson and Clarke [1982] synthesis method, $M, S_0 \models \mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$, since f is a conjunct of the specification, and M is produced by applying the Emerson and Clarke [1982] synthesis method to the specification. By Proposition 4.1.2.1, M and M' satisfy the same CTL formulae. Hence $M', S_0' \models \mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i q_i)$. Also, since M is in reachable form and M, M' are bisimilar (see the proof of Proposition 4.1.2.1), we conclude that M' is in reachable form.

It is easy to see that a global state s satisfies p_i, q_i iff its P_i-projection $s{\uparrow}i$ satisfies p_i, q_i respectively (since satisfaction of p_i, q_i depends only on the values assigned to atomic propositions in $\mathcal{AP}_i$, and $s, s{\uparrow}i$ agree, by definition, on these values). Now $P = P_1 \| \cdots \| P_K$ is extracted from M', and so, by the program extraction definition (2.3), every i-state s_i of P_i which satisfies p_i has as local successors in P_i i-states t_i which satisfy q_i. In Phase 2, an arc $(s_i, B \rightarrow //_{m \in [1:n]} A^m, t_i)$ of P_i is decomposed into arcs having the forms $(s_i, B \rightarrow skip, u_i^{m_0})$, $(u_i^{m_0}, true \rightarrow A^{m_1}, u_i^{m_1})$, ..., $(u_i^{m_{k-1}}, true \rightarrow A^{m_k}, u_i^{m_k})$, ..., $(u_i^{m_{n-1}}, true \rightarrow A^{m_n}, t_i)$. From the construction of Phase 2, (in particular, the way the local atomic proposition valuations of the "new" states $u_i^{m_0}, \dots, u_i^{m_{n-1}}$ are chosen) we see that each $u_i^{m_k}$, $k \in [0 : n-1]$, satisfies either p_i or q_i. Since t_i satisfies q_i, and s_i is chosen arbitrarily, we have that every i-state in P_i'' which satisfies p_i (namely, s_i and possibly some of the $u_i^{m_k}$) has as local successors in P_i'' i-states which either satisfy p_i or satisfy q_i (the i-states satisfying q_i being t_i and possibly some of the $u_i^{m_k}$).

Now M'' is the global-state transition diagram of $P'' = P_1'' \| \cdots \| P_K''$. Thus, by the global-state transition diagram definition (2.2.2) and CTL semantics, $M'', S'_0 \models \mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$. We can easily see, by CTL semantics, that removing transitions from M'' cannot cause $\mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$ to be violated. This is because AY_i is the weak nexttime operator. It does not actually require the existence of any transitions per se, but only requires that those transitions that leave a state satisfying p_i enter a state satisfying $p_i \vee q_i$. Since $S'''_0 \subseteq S'_0$ and $R''' \subseteq R''$ (by construction of Phase 4), we conclude $M''', S'''_0 \models \mathsf{AG}(p_i \Rightarrow \mathsf{AY}_i(p_i \vee q_i))$. □

B. GLOSSARY OF SYMBOLS

$\models$	Satisfies relation of CTL
$\{\mid\}$	State to formula operator
$\uparrow i$	State projection onto process i
$\downarrow i$	State projection onto all shared variables and all processes except i
M^1, M^2	Kripke structure
M, M', M'', M''', M^{iv}	The specific Kripke structures involved in the synthesis method
P, P', P'', P'''	The specific concurrent programs involved in the synthesis method
$R, R^1, R^2, R'', R''', R^{iv}$	Transition relation
S	Set of global states
S_0, S'_0, S'''_0	Set of initial global states
$\mathcal{AP}$	The set of atomic propositions
$\mathcal{AP}_i$	The set of atomic propositions of process i
$\mathcal{LO}$	Constructor of sets of propositional formulae
$\mathcal{SH}$	The set of shared variables

ACKNOWLEDGMENTS

We thank Amir Pnueli for suggesting the possibility of synthesizing Peterson's mutual exclusion algorithm, and Anish Arora for suggesting the barrier synchronization example.

REFERENCES

Anuchitanukul, A. and Manna, Z. 1994. Realizability and synthesis of reactive modules. In *Proceedings of the 6th International Conference on Computer Aided Verification.* Lecture Notes in Computer Science, vol. 818. Springer-Verlag, Berlin, 156–169.

Arora, A. 1992. A foundation of fault-tolerant computing. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin.

Arora, A., Attie, P. C., and Emerson, E. A. 1998. Synthesis of fault-tolerant concurrent systems. In *Seventeenth Annual ACM Symposium on Principles of Distributed Computing.* ACM Press, 1515 Broadway, New York, NY 10036, 173 – 182.

Attie, P. C. 1995. Formal methods for the synthesis of concurrent programs from temporal logic specifications. Ph.D. thesis, Dept. of Computer Sciences, The Univ. of Texas at Austin.

Attie, P. C. 1999. Synthesis of large concurrent programs via pairwise composition. In *CONCUR'99: 10th International Conference on Concurrency Theory.* Lecture Notes in Computer Science 1664. Springer-Verlag, Heidelberg, Germany.

Attie, P. C. and Emerson, E. A. 1998. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst. 20,* 1 (Jan.), 51–115. Preliminary version appears in *Sixteenth ACM Symposium on Principles of Programming Languages, 1989.*

Back, R. 1989. Refinement calculus, part ii: parallel and reactive programs. In *Stepwise refinement of distributed systems*, J. de Bakker, W. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science 430. Springer-Verlag, Heidelberg, Germany, 67–93.

Back, R. and von Wright, J. 1994. Trace refinement of action systems. In *CONCUR'94:Concurrency theory*, B. Johnsson and J. Parrow, Eds. Lecture Notes in Computer Science 836. Springer-Verlag, Heidelberg, Germany, 367–384.

Browne, M., Clarke, E. M., and Grumberg, O. 1988. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science 59*, 115–131.

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design*. Addison-Wesley, Reading, Mass.

Clarke, E. M., Grumberg, O., and Browne, M. C. 1986. Reasoning about networks with many identical finite-state processes. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 240 – 248.

Dijkstra, E. W. 1976. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, N.J.

Dill, D. and Wong-Toi, H. 1990. Synthesizing processes and schedulers from temporal specifications. In *International Conference on Computer-Aided Verification*. Number 531 in LNCS. Springer-Verlag, Heidelberg, Germany, 272–281.

Emerson, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. V. Leeuwen, Ed. Vol. B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, Mass.

Emerson, E. A. and Clarke, E. M. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program. 2*, 241 – 266.

Francez, N. 1986. *Fairness*. Springer-Verlag, New York.

Gribomont, E. 1990. Stepwise refinement and concurrency: the finite-state case. *Sci. Comput. Program. 14,* 2–3 (Oct.), 185–228.

Gribomont, E. 1993. Concurrency without toil: a systematic method for paralel program design. *Sci. Comput. Program. 21,* 1 (Aug.), 1–56.

Herlihy, M. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 11,* 1 (Jan.), 124–149.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12,* 10, 576–580, 583.

Lamport, L. 1990. A theorem on atomicity in distributed algorithms. *Distributed Computing 4,* 2, 59–68.

Lynch, N. and Tuttle, M. 1987. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6'th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 1515 Broadway, New York, NY 10036, 137 – 151.

Lynch, N. and Vaandrager, F. 1995. Forward and backward simulations — part I: Untimed systems. *Inform. and Comput. 121,* 2 (Sept.), 214–233.

Manna, Z. and Pnueli, A. 1995. *Temporal verification of reactive systems: safety*. Springer-verlag, New York.

Manna, Z. and Wolper, P. 1984. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst. 6,* 1 (Jan.), 68–93. Also appears in Proceedings of the Workshop on Logics of Programs, Yorktown-Heights, N.Y., Springer-Verlag Lecture Notes in Computer Science (1981).

Peterson, G. 1981. Myths about the mutual exclusion problem. *IPL 12,* 3 (June), 115–116.

Pnueli, A. and Rosner, R. 1989a. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. ACM, New York, 179–190.

Pnueli, A. and Rosner, R. 1989b. On the synthesis of asynchronous reactive modules. In *Proceedings of the 16th ICALP*. Lecture Notes in Computer Science, vol. 372. Springer-Verlag, Berlin, 652–671.

Singh, A., Anderson, J., and Gouda, M. 1994. The elusive atomic register. *J. ACM 41,* 2 (Mar.), 311–339.

Received July 1998; accepted June 1999