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

(Extended Abstract)

Paul C. ATTIE †
School of Computer Science
Florida International University
attie@fiu.edu

E. Allen EMERSON ‡
Department of Computer Sciences
The University of Texas at Austin
emerson@cs.utexas.edu

Abstract

Methods for mechanically synthesizing concurrent programs from temporal logic specifications have been proposed (cf. [EC82, MW84, PR89, PR89b, AM94]). An important advantage of these synthesis methods is that they obviate the need to manually construct a program and compose 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 (cf. [MW84]) or processes with global information about the system state (cf. [EC82]). Even simple synchronization protocols based on atomic read / atomic 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.

1 Introduction

Methods for synthesizing 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 [EC82] and Manna and Wolper [MW84]. 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 computation. For example, the method of Manna and Wolper [MW84] 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 [EC82] 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.

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. This method is an extension of the method of [EC82]. Essentially we first synthesize a correct program which, 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 may have been introduced by the decomposition. We illustrate our method by synthesizing an atomic read / atomic write solution for the mutual exclusion problem.

The paper is organized as follows: Section 2 defines the model of computation and the specification language. Section 3 presents the synthesis method, together with statements of some of the theorems that express the soundness of the method. Finally, section 4 presents our conclusions and discusses further work. For reasons of space, all proofs are omitted, and are provided in the full paper. Throughout, we use the mutual exclusion problem as a running example.

†Supported in part by Rome Laboratory, U. S. Air Force, under Contract No. F30602-93-C-0247.

‡Supported in part by NSF Grant CCR-9415496 and by SRC Contract DP 95-DP-388.

2 Preliminaries

2.1 Model of Parallel Computation

We consider concurrent programs of the form $P = P_1 \| \cdots \| P_K$ which 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), and each arc is labeled with a synchronization command $B \to A$ consisting of an enabling condition (i.e., guard) B and corresponding action A to be performed (i.e., a guarded command [Dij76]). Self-loops, where there is an arc from a node to itself, are disallowed. A *global state* is a tuple of the form $(s_1, \ldots, s_K, x_1, \ldots, x_m)$ where each node s_i is the current local state of P_i and $x_1, \ldots, x_m$ is a list (possibly empty) of shared synchronization variables. A guard B is a predicate on global states and an action A is a parallel assignment statement which 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 parallelism in the usual way by the nondeterministic interleaving of the "atomic" transitions of the individual synchronization skeletons of the processes P_i. Hence, at each step of the computation, some process with an enabled transition is nondeterministically selected to be executed next. Assume that the current state is $s = (s_1, \ldots, s_i, \ldots, s_K, x_1, \ldots, x_m)$ and that process P_i contains an arc from node s_i to s_i' labeled by the command $B \to A$ (such a P_i-*arc* will be written as $(s_i, B \to A, s_i')$). If B is true in the current state then a permissible next state is $s' = (s_1, \ldots, s_i', \ldots, s_K, x_1', \ldots, x_m')$ where $x_1', \ldots, x_m'$ is the list of updated shared variables resulting from action A (we notate this transition as $s \xrightarrow{i,A} s'$). A computation path is any sequence of states where each successive pair of states is related by the above next state relation.

The synthesis task thus amounts to supplying the commands to label the arcs of each process' synchronization skeleton so that the resulting computation trees of the entire program $P_1 \| \cdots \| P_K$ meet a given Temporal Logic specification.

2.2 The Specification Language CTL

Our specification language is the propositional branching time temporal logic CTL [EC82]. We have the following syntax for CTL, where p denotes an atomic proposition, and f, g denote (sub-)formulae. The atomic propositions are drawn from a set $\mathcal{AP}$ that is partitioned into sets $\mathcal{AP}_1, \ldots, \mathcal{AP}_K$. $\mathcal{AP}_i$ contains the atomic propositions local to process i.

- Each of p, $f \wedge g$ and $\neg f$ is a formula (where the latter two constructs indicate conjunction and negation, respectively).
- $EX_j f$ is a formula which means that there is an immediate successor state reachable by executing one step of process P_j in which formula f holds.
- $A[fUg]$ is a formula which means that for every computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.
- $E[fUg]$ is a formula which means that for some computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.

Formally, we define the semantics of CTL formulae with respect to a Kripke structure $M = (S_0, S, R)$ consisting of a countable set S of global states, a set $S_0 \subseteq S$ of initial states, and a binary transition relation $R \subseteq S \times S$, giving the transitions of every process. R is partitioned into relations $R_1, \ldots, R_K$, where R_i gives the transitions of process i. We label each state s with the set $V[s] \subseteq \mathcal{AP}$ of atomic propositions that are true in s, and we label each transition with a parallel assignment statement (if no variables are altered, then we label the transition with *skip*). We require that R be total, i.e., $\forall x \in S,\ \exists y : (x, y) \in R$. A *fullpath* is an infinite sequence of states $(s_0, s_1, s_2 \ldots)$ such that $\forall i : (s_i, s_{i+1}) \in R$. 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:

$$
\begin{array}{ll}
M, s_0 \models p & \text{iff } p \in V(s_0) \\
M, s_0 \models \neg f & \text{iff not}(s_0 \models f) \\
M, s_0 \models f \wedge g & \text{iff } s_0 \models f \text{ and } s_0 \models g \\
M, s_0 \models EX_j f & \text{iff for some state } t, \\
 & \quad (s_0, t) \in R_j \text{ and } t \models f, \\
M, s_0 \models A[fUg] & \text{iff for all fullpaths } (s_0, s_1, \ldots) \text{ in } M, \\
 & \quad \exists i[\ i \geq 0 \text{ and } s_i \models g \text{ and} \\
 & \quad\quad \forall j(0 \leq j \wedge j < i \Rightarrow s_j \models f)\ ] \\
M, s_0 \models E[fUg] & \text{iff for some fullpath } (s_0, s_1, \ldots) \text{ in } M, \\
 & \quad \exists i[\ i \geq 0 \text{ and } s_i \models g \text{ and} \\
 & \quad\quad \forall j(0 \leq j \wedge j < i \Rightarrow s_j \models f)\ ]
\end{array}
$$

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 $(f \Rightarrow g) \wedge (g \Rightarrow f)$, AFf for $A[trueUf]$, EFf for $E[trueUf]$, AGf for $\neg EF \neg f$, EGf for $\neg AF \neg f$, $AX_i f$ for $\neg EX_i \neg f$, EXf for $EX_1 f \vee \cdots \vee EX_k f$, and AXf for $AX_1 f \wedge \cdots \wedge AX_k f$. The two process mutual exclusion problem is specified in CTL as follows:

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

(1) 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:
$AG(N_1 \Rightarrow (AX_1 T_1 \wedge EX_1 T_1))$
$AG(N_2 \Rightarrow (AX_2 T_2 \wedge EX_2 T_2))$

(2) It is always the case that any move P_1 makes from its Trying region is into its Critical region. Likewise for P_2:
$AG(T_1 \Rightarrow AX_1 C_1)$
$AG(T_2 \Rightarrow AX_2 C_2)$

(3) 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:
$AG(C_1 \Rightarrow (AX_1 N_1 \wedge EX_1 N_1))$
$AG(C_2 \Rightarrow (AX_2 N_2 \wedge EX_2 N_2))$

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

(5) P_1, P_2 do not starve:
$AG(T_1 \Rightarrow AFC_1)$
$AG(T_2 \Rightarrow AFC_2)$

(6) P_1, P_2 do not access critical resources together:
$AG(\neg(C_1 \wedge C_2))$

(7) It is always the case that some process can move:
$AGEXtrue$

2.3 Technical Definitions

For a global state $s = \{s_1, \ldots, s_K, x_1, \ldots, x_m\}$, let $s{\Uparrow}i \stackrel{\mathrm{df}}{=} s_i$. We extend the labeling function V to local states as follows: $V[s_i] = V[s] \cap \mathcal{AP}_i$. For $Q_i \in \mathcal{AP}_i$, we write $s_i(Q_i) = true$, $s(Q_i) = true$ iff $Q_i \in V[s_i]$, $Q_i \in V[s]$ respectively, and $s_i(Q_i) = false$, $s(Q_i) = false$ otherwise, respectively. $s{\Downarrow}i \stackrel{\mathrm{df}}{=} s - \{s_i\}$, i.e., $s{\Downarrow}i$ is s with its P_i-component removed. $\mathcal{SH}$ denotes the set $\{x_1, \ldots, x_m\}$ of shared variables. $\{\!\{s\}\!\} \stackrel{\mathrm{df}}{=}$ "$(\bigwedge_{s(Q)\,=\,true} Q) \wedge (\bigwedge_{s(Q)\,=\,false} \neg Q) \wedge (\bigwedge_{x\,\in\,\mathcal{SH}} x = s(x))$" where Q ranges over $\mathcal{AP}$, and $\{\!\{s{\Downarrow}i\}\!\} \stackrel{\mathrm{df}}{=}$ "$(\bigwedge_{s(Q)\,=\,true} Q) \wedge (\bigwedge_{s(Q)\,=\,false} \neg Q) \wedge (\bigwedge_{x\,\in\,\mathcal{SH}} x = s(x))$," where Q ranges over $\mathcal{AP} - \mathcal{AP}_i$. $\{\!\{s\}\!\}$ 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.

A particular P_i-arc $(s_i, B \to A, t_i)$ generates a set of P_i-transitions; one for every state s such that $s{\Uparrow}i = s_i$ and $s(B) = true$. We call such a set a *family*, and use the term P_i-family to specify a family generated by an arc of process P_i. Formally, 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 having 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 $t{\Uparrow}i = t'{\Uparrow}i$. If $s \xrightarrow{i,A} t \in \mathcal{F}$, then let $\mathcal{F}.start$, $\mathcal{F}.finish$, $\mathcal{F}.assig$ denote $s{\Uparrow}i$, $t{\Uparrow}i$, and 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 $\bigvee_{T\,\in\,\mathcal{F}} \{\!\{(T.begin){\Downarrow}i\}\!\}$.

Definition 1 (*Program Extraction*)
Let $M = (S_0, S, R)$ be an arbitrary Kripke structure. Then the program $P = P_1 \| \cdots \| P_K$ is extracted from M as follows. For all $i \in \{1, \ldots, K\}$:

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

Two i-states are *propositionally equivalent* ($\sim$) iff they have the same labels: $s_i \sim t_i$ iff $V[s_i] = V[t_i]$. We use $\doteq$ to denote syntactic equality, i.e., $B \doteq true$ means that B is the constant $true$. If B is $N_1 \vee \neg N_1$, for example, then $B \not\doteq true$. An arc $(s_i, B \to A, t_i)$ is *guarded* iff $B \not\doteq true$, and *unguarded* iff $B \doteq true$. An arc $(s_i, B \to A, t_i)$ is *single-writing* iff $A \doteq skip$ and $s{\Uparrow}i \not\sim t{\Uparrow}i$, or $A \doteq$ "$x := c$" (for some $x \in \mathcal{SH}$ and constant c) and $s{\Uparrow}i \sim t{\Uparrow}i$. An arc $(s_i, B \to /\!/_{m\in[1:n]}\ x^m := c^m$ [§]$, t_i)$ is *multiple-writing* iff $(n > 0$ and $s{\Uparrow}i \not\sim t{\Uparrow}i)$

§ $/\!/_{m\in[1:n]}\ x^m := c^m$ is a parallel assignment statement: the assignments $x^m := c^m$ are executed simultaneously. The c^m are all constants. Henceforth, we use "assignment" instead of "parallel assignment statement."

or $n > 1$. An arc is *writing* iff it is either single-writing or multiple-writing. An arc $(s_i, B \rightarrow skip, t_i)$ is *non-writing* iff $s_i \sim t_i$. An arc is *test-and-set* iff it is both guarded and writing. We extend these attributes to families as follows. A family $\mathcal{F}$ is single-writing, multiple-writing, writing, non-writing iff the arc $(\mathcal{F}.start, \mathcal{F}.guard \rightarrow \mathcal{F}.assig, \mathcal{F}.finish)$ is single-writing, multiple-writing, writing, non-writing respectively. A P_i-family $\mathcal{F}$ is *unguarded in* $M = (S_0, S, R)$ iff for all reachable states s in M such that $s{\uparrow}i = \mathcal{F}.start$: $s \xrightarrow{i,A} t \in R$. Here $A = \mathcal{F}.assig$, and t is some global state such that $t{\uparrow}i = \mathcal{F}.finish$. If $\mathcal{F}$ is not unguarded in M, then we say that $\mathcal{F}$ is *guarded in* M.

3 The Synthesis Method

Let f be a specification, expressed in CTL, for a concurrent program. We proceed as follows. We first apply the CTL decision procedure to f as in [EC82]. 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 which satisfies f, and P can be extracted from M via definition 1. 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 which violate the program specification. We address this by generating the global state transition diagram of the decomposed program, and then deleting all the portions of this diagram which are inconsistent with the specification. If some initial state is not deleted, then, from the resulting structure, an atomic read / atomic write program that satisfies the specification can be extracted. Our method consists of a sequence of phases. Each phase takes as input the output of the previous phase:

Phase 1 Produce an initial program which satisfies the specification but is not necessarily in atomic read / atomic write form.

Phase 2 Decompose the initial program into an atomic read / atomic write program

Phase 3 Delete portions of the global state transition diagram (of the program produced in phase 2) that violate the specification

Phase 4 Extract a final atomic read / atomic write program that satisfies the specification

3.1 Phase 1: Produce an Initial Program

First, we apply the CTL decision procedure to the program specification f, obtaining a model M of f (if f is satisfiable). Figure 1 shows M for the mutual exclusion specification given above. Second, we transform $M = (S_0, S, R)$ into an "equivalent" Kripke structure $M' = (S_0', S, R')$ where every assignment $//_{m\in[1:n]}\ x^m := c^m$ (executed by some process P_i) in M is replicated along all "compatible" transitions. This has the desirable effect of weakening the guard of $//_{m\in[1:n]}\ x^m := c^m$ in the skeletons extracted from M'. Let $//_{m\in[1:n]}\ x^m := c^m$ be a assignment which 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: 1) $u{\uparrow}i = s{\uparrow}i, v{\uparrow}i = t{\uparrow}i$, (i.e., every P_i-transition which takes P_i from the same (local) start state to the same (local) finish state), and, 2) A does not assign to any of $x^1, \ldots, x^n$. After this replication is performed (for all assignments in M), it is possible that some states in M may have two different values for the same shared variable due to the extra assignments introduced. We therefore apply the following "propagation rules" repeatedly to M, until none of the rules produces any change. M' is the resulting structure.

add-prop If a transition into state s is labeled with $x := c$ then add the proposition $x = c$ to s

split-state If state s contains propositions $x = c^1, \ldots, x = c^k$ $(k > 1)$ then replace s by k states $s^1, \ldots, s^k$, where s^ℓ contains $x = c^\ell$ and all propositions 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 which are consistent with $x = c^\ell$.

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

These rules resolve inconsistencies arising from the replication of assignments by creating new global states. The rules do not introduce new cycles, or states that are propositionally different from every reachable state in M. We let S_0', the set of initial states of M', be $\{s \mid s$ is a state of M' and s is propositionally equivalent to some state in $S_0\}$. It is straightforward to establish a bisimulation [CGB86] between M and M': every state s in M is mapped to all states in M' that resulted from s being split (if s was not split, then s is mapped to "itself" in M'). Since the **split-state** rule preserves all outgoing transitions, i.e., all successor states, it follows (by a simple induction) that the mapping given above is indeed a bisimulation. Hence, by theorem 2 of [CGB86], the two structures satisfy the same CTL formulae. Thus the structure $M' = (S'_0, S, R')$ produced satisfies all of the specifications that M does.

Third, we extract the program $P = P_1 \| \cdots \| P_K$ from M' using definition 1. Finally, we simplify the guards in the skeletons for $P_1, \ldots, P_K$ as follows. Consider an arbitrary arc $(s_i, B \rightarrow A, t_i)$ of P_i. By definition 1, B is in disjunctive normal form. Let $b_1 \wedge \ldots \wedge b_n$ be a disjunct of B. If $\{s_i\} \wedge (\bigwedge_{j \in [1:n]-\{k\}} b_j) \Rightarrow b_k$ holds in all reachable states of M' for some k ($k \in [1:n]$), then we can eliminate b_k from $b_1 \wedge \ldots \wedge b_n$. Figure 2 illustrates the resulting program (derived from figure 1). The guard of the assignment operation $x := 2$ is "*true*" here, whereas in the program extracted directly from figure 1, the guard of $x := 2$ would be T_2.

3.2 Phase 2: Decompose the Initial Program

First, we "group" all the atomic propositions in $\mathcal{AP}_i$ into a single variable $L_i \subseteq \mathcal{AP}_i$, the *externally visible location counter*. $s_i(L_i)$, the value of L_i in s_i, is simply $V[s_i]$, the label of s_i (i.e., the set of atomic propositions in $\mathcal{AP}_i$ that are true in s_i). We replace all references to atomic propositions by means of the following transformations.

- Every occurrence of an atomic proposition $Q_i \in \mathcal{AP}_i$ in some guard is replaced by "$Q_i \in L_i$"
- Every arc $(s_i, B \rightarrow A, t_i)$ of P_i such that $V[s_i] \neq V[t_i]$, is replaced by the arc $(s_i, B \rightarrow A // L_i := V[t_i], t_i)$

If the atomic propositions in $\mathcal{AP}_i$ are mutually exclusive and exhaustive (as in our mutual exclusion example), then $s_i(L_i)$ is a singleton, say $\{Q_i\}$. We write $s_i(L_i) = Q_i$ in this case. Also, $L_i := V[s_i], Q_i \in L_i$ are written $L_i := Q_i, L_i = Q_i$ respectively. Figure 3 shows the program of figure 2 after the externally visible location counters (henceforth referred to simply as location counters) L_1 and L_2 have been introduced.

Next, we decompose every test-and-set arc into a guarded and non-writing arc (for the "test"), followed by an unguarded and writing arc (for the "set"). Finally, we replace every (unguarded and) multiple-writing arc by a set of sequences of unguarded and single-writing arcs. Each sequence represents one order of serialization of the write operations of the original multiple-writing arc. This decomposition may introduce several local states with the same propositional labeling into a skeleton. We therefore assign to every local state s_i an integer value num_i which serves to distinguish it from other identically labeled local states. This value is shown as a superscript in all figures.

Let $P'' = P''_1 \| \cdots \| P''_K$ be the resulting program. Figure 4 shows P'' for the mutual exclusion example. 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 $L_1 := T_1$.

Proposition 1 *Every arc in the skeletons of P'' is either guarded and non-writing, or unguarded and single-writing.*

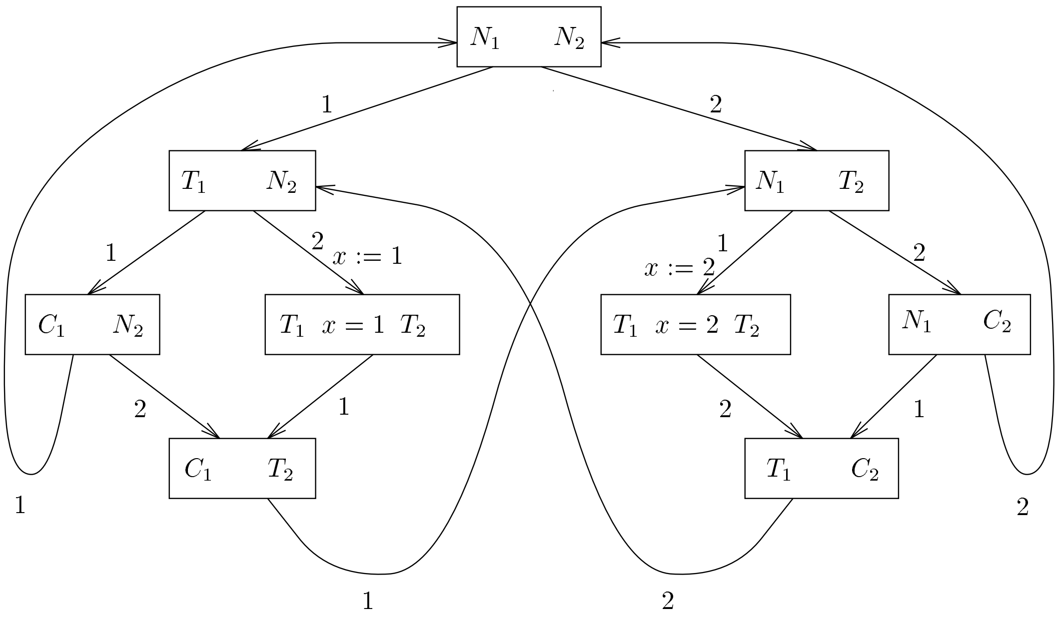
3.3 Phase 3: Delete Portions of the Global State Transition Diagram that Violate the Specification

First, we generate the global state transition diagram $M'' = (S'_0, S, R'')$ of P''. Second, 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 as follows (in the sequel, h is a purely propositional formula over $\mathcal{AP}$, and p_i, q_i are purely propositional formulae over $\mathcal{AP}_i$):

For each conjunct f of the specification, except those of the form $AG(p_i \Rightarrow AX_i q_i)$, label the reachable states of M'' according to the following rules.

- If $f = h$, then label every initial state in M'' (i.e., every state in S'_0) with h
- If $f = AGh$, then label every reachable state in M'' with h
- If $f = AG(p_i \Rightarrow AFq_i)$ then label every reachable state s in M'' such that $s \models p_i$ with AFq_i
- If $f = AG(p_i \Rightarrow EX_i q_i)$ then label every reachable state s in M'' such that $s \models p_i$ with $EX_i(p_i \vee q_i)$

Third, we check that every reachable state of M'' satisfies all the formulae in its label. If some state of 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. These deletions are performed according to a set of *deletion rules*, shown below. In principle, we can formulate deletion rules for full CTL, cf. [EC82]. For simplicity, we show deletion rules for the fragment of CTL used to specify the mutual exclusion problem. Now deleting a transition T_i (of some process P''_i of P'') causes the arc AR_i corresponding to the family containing T_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 T_i, thereby deleting AR_i entirely. However, this may leave P''_i incapable of infinite behavior, i.e., 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



The initial state set is { $[N_1\ N_2]$ }

Figure 1: Model of the Mutual Exclusion Specification Produced by the CTL Decision Procedure

delete transitions in M'' that correspond to deletable arcs. A transition T_i is deleted by invoking the procedure $delete(T_i)$:

procedure $delete(T_i)$

Let $\mathcal{F}_i$ be the family in M'' containing T_i, and let AR_i be the skeleton arc corresponding to $\mathcal{F}_i$.
if $\mathcal{F}_i$ is guarded (and non-writing) **then**
  remove T_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

Note that, since M'' is generated from P'', we have, by proposition 1, that every family in M'' is either guarded and non-writing, or unguarded and single-writing.

The deletion rules are as follows. 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.

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 T_i from every initialized path ending in s, and remove T_i from M'' by invoking $delete(T_i)$ (an initialized path is a path starting in an initial state).

AF-rule $AFq_i \in label(s)$ and $s \not\models AFq_i$:
Find one deletable transition T_i from every fullpath π starting in s such that $\pi \not\models Fq_i$, and remove T_i from M'' by invoking $delete(T_i)$.

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

EX-rule $s \not\models EXtrue$, 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 $(s_i, B \to A, t_i)$ is an arc in P''_i such that either P''_i contains no arc with start state t_i, or (P''_i contains no arc with finish state s_i, and $s_i \notin S'_0{\uparrow}i$):
Remove $(s_i, B \to A, t_i)$ from P''_i, and its corresponding family from M''.

For all the above rules, whenever a state s in S'_0 is deleted, all transitions in M'' 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.

The deletion rules are applied as long as possible. Since M'' is finite, and each application of a deletion rule

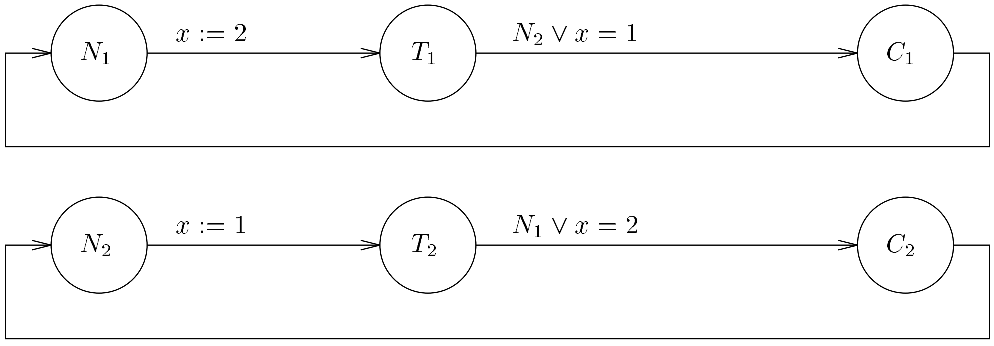


Figure 2: Mutual exclusion program derived from figure 1

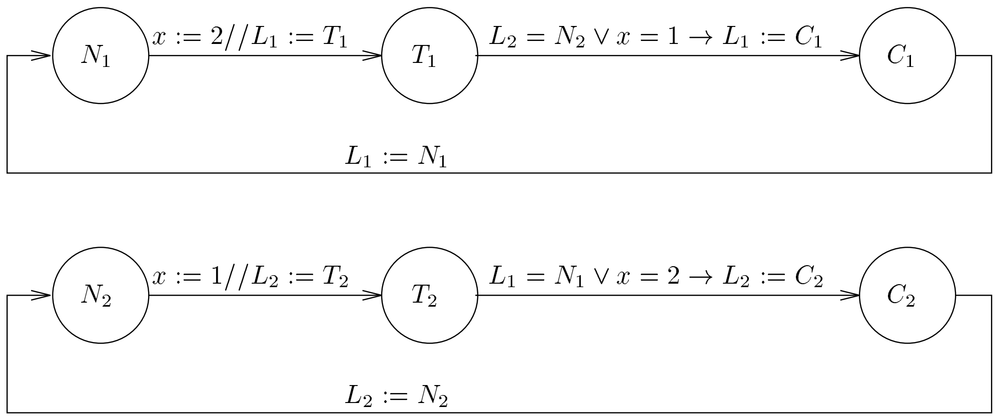


Figure 3: Mutual exclusion program after introduction of the externally visible location counters

results in the deletion of at least one state or one transition in M'', we eventually terminate. 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 structure from M'' which 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 / atomic write program satisfying a given specification, even if such a program does in fact exist. The method is sound however, as we shall subsequently establish. If S_0''' is nonempty, we let M''' be the structure (S_0''', S, R'''). We show below that M''' satisfies the given specification.

Figure 5 shows M''' for our running example. The assignments to L_1, L_2 are omitted since they are easily inferred from the begin and end states of each transition. Note that the global state transition diagram of the program of figure 4 contains the path $[N_1^1\,1\,N_2^1] \xrightarrow{2,x:=1} [N_1^1\,1\,N_2^2] \xrightarrow{1,x:=2} [N_1^2\,2\,N_2^2] \xrightarrow{1} [T_1^2\,2\,N_2^2] \xrightarrow{1} [T_1^3\,2\,N_2^2] \xrightarrow{2} [T_1^3\,2\,T_2^2] \xrightarrow{2} [T_1^3\,2\,T_2^3] \xrightarrow{1} [C_1^1\,2\,T_2^3] \xrightarrow{2} [C_1^1\,2\,C_2^1]$ that ends in a state which violates the mutual exclusion constraint $AG(\neg(C_1 \wedge C_2))$. This path (and all such paths) has been eliminated by application of the deletion rules.

Proposition 2 *If $S_0''' \neq \emptyset$, and f is a conjunct of the specification, then $M''', S_0''' \models f^*$*

where $f^* = f$ if f has one of the forms h, AGh, $AG(p_i \Rightarrow AFq_i)$, and $f^* = AG(p_i \Rightarrow EX_i(p_i \vee q_i))$, $AG(p_i \Rightarrow AX_i(p_i \vee q_i))$ if $f = AG(p_i \Rightarrow EX_iq_i)$, $AG(p_i \Rightarrow AX_iq_i)$ respectively.

3.4 Phase 4: Extract the Final Program

A *simple term* has the form $Q_i \in L_i$, or the form $x = c$. In an atomic read / atomic 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. Likewise, a disjunction of simple terms can be used as the guard of an arc since checking

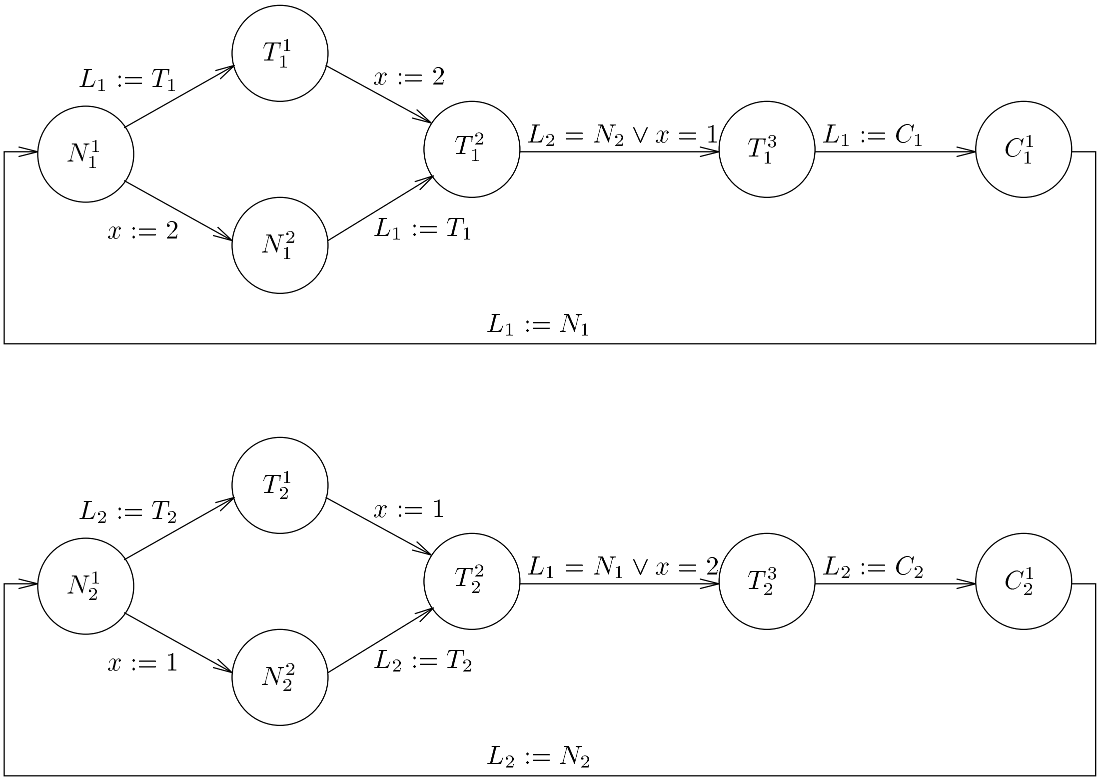


Figure 4: Decomposed mutual exclusion program

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* entails checking that all conjuncts evaluate to *true* simultaneously. This requires the simultaneous reading of all shared variables and location counters referenced in any conjunct, and the atomic read / atomic write model does not permit such "multiple" reads. We must therefore extract the final program in a manner which ensures that all guards are disjunctions of simple terms. We proceed as follows.

For every guarded P_i-family $\mathcal{F}$ in M''', we find 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

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

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

where $\mathcal{F}_k^r$ is the set of all reachable transitions in $\mathcal{F}_k$. In other words, b_k is satisfied by all of the states that are the begin state of some transition in $\mathcal{F}_k^r$, and is not satisfied by any state whose P_i-projection is the start state of $\mathcal{F}$, and 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 all transitions in $\mathcal{F}_k$. Hence, the guard $\bigvee_{k \in [1:n]} b_k$ takes into account all 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 s is not the start state of some transition in $\mathcal{F}_k$, ensures that no "extra" transitions are generated by the extracted program (i.e., no transitions that are not present in the Kripke structure M''' 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.

If, for some $\mathcal{F}_k$, no b_k can be found for which (G1), (G2) hold, then we attempt to make (G1), (G2) true for some b_k by deleting reachable states which violate (G1) or (G2) (or both). Since such deletions may, in general, cause violation of the specification, we must repeat phase 3 after one or more of these deletions are performed.

Once the guard for every guarded family has been computed, we extract the final atomic read / atomic write program $P''' = P_1''' \| \cdots \| P_K'''$ from M''' according to the following definition.

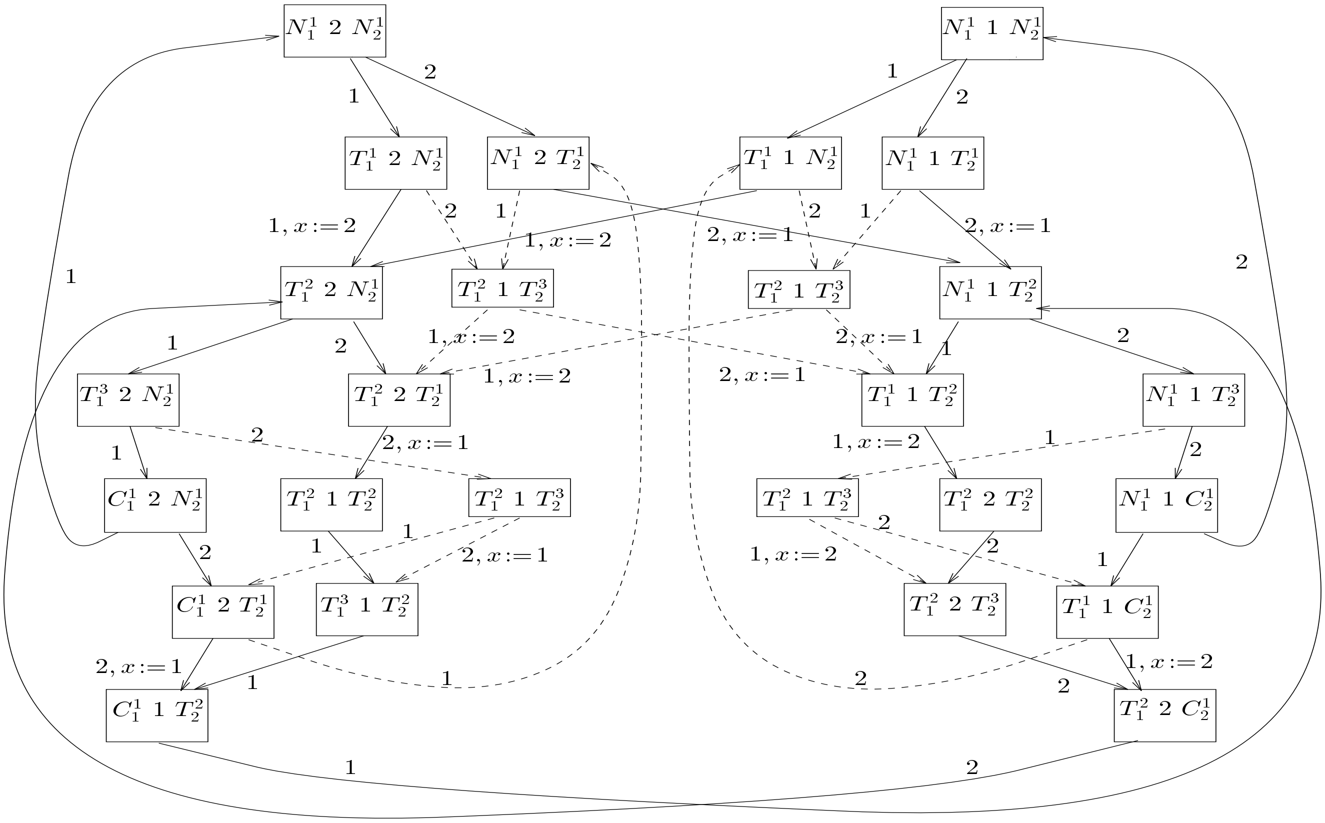


Figure 5: Global state transition diagram of the program of figure 4 after all deletions have been performed

Definition 2 (*Atomic Read / Atomic Write Program Extraction*)
The program $P''' = P_1'''\| \cdots \|P_K'''$ is extracted from M''' as follows. For all $i \in \{1, \ldots, K\}$:

$(s_i, B \to skip, t_i) \in P_i'''$ *iff*
there exists a guarded and non-writing P_i-family $\mathcal{F}$ in M''' such that
$s_i = \mathcal{F}.start$, $t_i = \mathcal{F}.finish$, and B is the guard computed for $\mathcal{F}$ above

$(s_i, true \to A, t_i) \in P_i'''$ *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 $A = \mathcal{F}.assig$

Consider figure 5, which gives M''' for our running example. There is exactly one guarded P_1-family $\mathcal{F}$ in M''', which contains 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 set $\mathcal{F}_1 = \{ [T_1^2\ 2\ N_2^1] \xrightarrow{1} [T_1^3\ 2\ N_2^1] \}$, and $\mathcal{F}_2 = \{ [T_1^2\ 1\ T_2^2] \xrightarrow{1} [T_1^3\ 1\ T_2^2] \}$. For $\mathcal{F}_1$, we find that $b_1 \doteq$ "$N_2 \in L_2$" satisfies (G1, G2), and for $\mathcal{F}_2$, we find that $b_2 \doteq$ "$x = 1$" satisfies (G1, G2). Thus a suitable guard for the arc corresponding to $\mathcal{F}$ is $N_2 \in L_2 \vee x = 1$. Thus the guarded arc $(T_1^2, N_2 \in L_2 \vee x = 1 \to skip, T_1^3)$ is extracted. The remaining arcs of P_1 in figure 6 are all extracted from unguarded families in a straightforward manner using definition 2. P_2 is extracted from figure 5 in a similar manner. The resulting program, shown in figure 6, is essentially Petersons solution [Pe81] to the mutual exclusion problem.

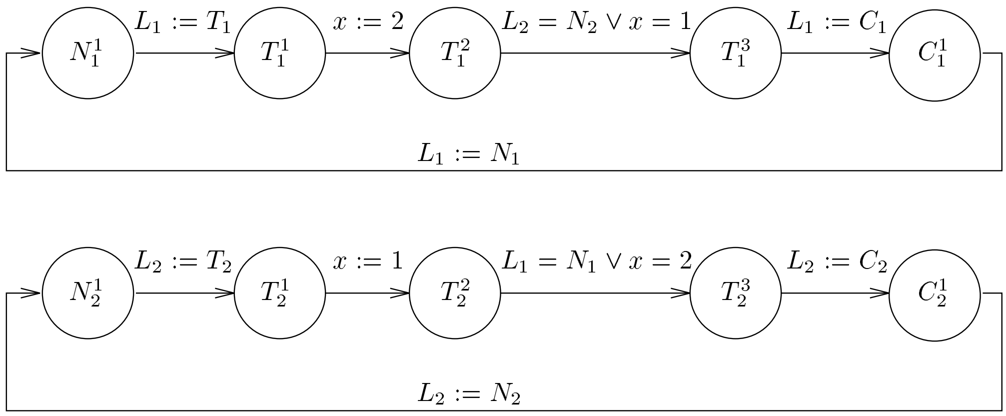
An arc $(s_i, B \to A, t_i)$ is *single-reading* iff B is a disjunction of simple terms.

Proposition 3 *Every arc in P''' is either single-reading and non-writing, or unguarded and single-writing.*

Lemma 4 (*Correct Extraction Lemma*)
Let $M^{iv} = (S_0''', S, R^{iv})$ be the global state transition diagram of P''' (with initial states S_0'''). If a state s is reachable in both M''', M^{iv}, then, for all i, A, t:
$$s \xrightarrow{i,A} t \in R''' \text{ iff } s \xrightarrow{i,A} t \in R^{iv}$$

Proposition 5 ($M''' - M^{iv}$ *Equivalence Proposition*)
Let f be an arbitrary formula of CTL. Then
$$M''', S_0''' \models f \text{ iff } M^{iv}, S_0''' \models f$$

Theorem 6 *If $S_0''' \neq \emptyset$, then P''' is an atomic read / atomic write program that satisfies the given CTL specification.*



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

Figure 6: Atomic Read / Atomic Write program for the two-process mutual exclusion problem

4 Conclusions and Further Work

We have presented a method for the synthesis of atomic read / atomic 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 3, there are, in general, many choices of transitions to be deleted. Thus the method is 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.

The main shortcoming of the method is its incompleteness. 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. We have also used the method to synthesize an atomic read / atomic write solution to the *mutual inclusion* problem [Ho86]. Finally, we note that the integer superscript introduced to distinguish propositionally identical local states essentially represents a hidden component of the location counter of each process that is not visible to other processes. Thus its introduction does not violate the atomic read / atomic write model.

Acknowledgments: We wish to thank Amir Pnueli for suggesting the possibility of synthesizing Peterson's solution.

References

[AM94] A. Anuchitanukul, Z. Manna: "Realizability and Synthesis of Reactive Modules," *CAV94*, Springer LNCS 818, (1994), 156–169.

[CGB86] E.M. Clarke, O. Grumberg, and M.C. Browne: "Reasoning About Networks With Many Identical Finite-State Processes," *Proc. 5'th ACM PODC*, (1986), 240–248.

[Dij76] E.W. Dijkstra: **A Discipline of Programming**, Prentice-Hall Inc., 1976.

[EC82] E.A. Emerson, E.M. Clarke: "Using Branching Time Temporal Logic To Synthesize Synchronization Skeletons," *Science of Computer Programming* 2 (1982) 241–266.

[Ho86] R. Hoogerwoord: "An Implementation of Mutual Inclusion," *IPL*, 23 (1986) 77–80.

[MW84] Z. Manna, P. Wolper: "Synthesis of Communicating Processes from Temporal Logic Specifications," *ACM TOPLAS*, 6 (1984) 68–93.

[Pe81] G.L. Peterson: "Myths About the Mutual Exclusion Problem," *IPL*, 12 (1981) 115–116.

[PR89] A. Pnueli, R. Rosner: "On the Synthesis of a Reactive Module," *Proc. 16'th ACM POPL*, (1989), 179–190.

[PR89b] A. Pnueli, R. Rosner: "On the Synthesis of Asynchronous Reactive Modules," *Proc. 16th ICALP*, Springer LNCS 372, (1989), 652–671.