On the Implementation Complexity of Specifications of Concurrent Programs

Paul C. Attie [1]

College of Computer Science, Northeastern University, Boston, MA
MIT Laboratory for Computer Science, Cambridge, MA
attie@ccs.neu.edu
http://www.ccs.neu.edu/home/attie/Attie.html

Abstract. We present a decision algorithm for the following problem: given a specification, does there exist a concurrent program which both satisfies the specification and which can be implemented in hardware-available operations in a straightforward manner, i.e, without long correctness proofs, and without introducing excessive blocking and/or centralization? In case our decision algorithm answers "yes," we also present a synthesis method to produce such a program. We consider specifications expressed in branching time temporal logic. Our result gives a way of classifying specifications as either "easy to implement" or "difficult to implement," and can be regarded as the first step towards a notion of "implementation complexity" of specifications.

1 Introduction

One of the major approaches to the construction of correct concurrent programs is *successive refinement*: start with a high-level specification, and construct a series of programs, each of which "refines" the previous one in some way. In the realm of shared-memory concurrent programs, this refinement usually takes the form of reducing the *grain of atomicity* of the operations used for inter-process communication and synchronization. For example, a high-level design might assume that the entire global state can be read and updated in a single atomic transition, whilst a low-level implementation would be restricted to the operations typically available in hardware: atomic reads and writes of registers, test-and-set of a single bit, load-linked/store-conditional, compare-and-swap, etc. Each of the successive refinements is considered correct if and only if it conforms to the specification. The notions of conformance to a specification which are widely studied can be roughly categorized into two approaches:

1. The use of an operational specification, e.g., an automaton or a labeled transition system, which is successively refined, via several intermediate levels of abstraction, into an implementation. The implementation is considered correct if and only if each of its externally visible behaviors ("traces") is also a trace of the specification, or if it is "bisimilar" to the specification.

[1] Supported in part by NSF Grant CCR-0204432

2. The use of a temporal logic formula as a specification. The program is considered correct iff its "semantic denotation" satisfies the formula. In the branching-time paradigm, the semantic denotation of a program is its global-state transition diagram, which can be viewed as a model-theoretic structure for a suitable branching-time temporal logic. The implementation is correct if and only if the specification is true in each of the initial states of the implementation. In the linear-time paradigm, the semantic denotation of a program is the set of its executions. Each execution can be viewed as a model-theoretic structure for a suitable linear-time temporal logic. The implementation is correct if and only if the specification is true along every execution.

We consider the following question: given a specification, does there exist a concurrent program which both satisfies the specification and which can be easily refined to hardware-available operations in a straightforward and efficient manner, i.e, without long correctness proofs, and without introducing excessive blocking and/or centralization? We use the branching-time temporal logic CTL [9, 10] to express specifications. For CTL specifications, we present an algorithm which decides this question in the sense that it detects a condition, temporary stability of action guards, which allows for easy refinement. When this condition holds, we provide a method of mechanically *synthesizing* a program which satisfies the specification and which can be easily refined.

Related work. Previous synthesis methods [2, 8, 10, 13–15, 17, 18] all produce high-grain concurrent programs. In [10], every process can read and update the global state in a single atomic transition. In [15], the synthesized program consists of a central "synchronizer" process which communicates with satellite processes, who do not communicate amongst each other. The methods of [2, 8, 13, 14, 17, 18] all synthesize a single "reactive module," which communicates with the environment. Thus, all these methods produce a centralized system consisting of a single process.

The rest of the paper is as follows. Section 2 presents technical preliminaries: our model of concurrent computation, and the specification language CTL. Section 3 gives some technical background on the CTL decision procedure. Section 4 presents our result: a decision procedure for answering the question posed above, and a synthesis method for the case when the answer is positive. Section 5 applies our result to the mutual exclusion and readers-writers problems. Section 6 discusses further work and concludes.

2 Technical Preliminaries

2.1 Model of Concurrent Computation

We consider concurrent programs of the form $P = P_1 \| \cdots \| P_I$ that consist of a finite number I of fixed sequential processes $P_1, \ldots, P_I$ running in parallel. With every process P_i, $1 \le i \le I$, we associate a single unique index i. Each P_i is a *synchronization skeleton* [10], that is, a state-machine where each (local) state of P_i represents a region of code intended to perform some sequential computation and where each arc represents a conditional transition (between different regions of sequential code) used to enforce synchronization constraints.

Formally, each P_i is a directed graph where each node is a (local) state of P_i and is labeled by a unique name (s_i), and where each arc is labeled with a guarded command [7] $B_i \to A_i$ consisting of a guard B_i and corresponding action A_i. With each P_i we associate a set $\mathcal{A}P_i$ of *atomic propositions*, and a mapping V_i from local states of P_i to subsets of $\mathcal{A}P_i$: $V_i(s_i)$ is the set of atomic propositions that are true in s_i. As P_i executes transitions and changes its local state, the atomic propositions in $\mathcal{A}P_i$ are updated. Different local states of P_i have different truth assignments: $V_i(s_i) \neq V_i(t_i)$ for $s_i \neq t_i$. Atomic propositions are not shared: $\mathcal{A}P_i \cap \mathcal{A}P_j = \emptyset$ when $i \neq j$. Other processes can read (via guards) but not update the atomic propositions in $\mathcal{A}P_i$. We define $\mathcal{A}P = \mathcal{A}P_1 \cup \cdots \cup \mathcal{A}P_I$. There is also a set of shared variables $x_1, \ldots, x_m$, which can be read and written by every process. These are updated by the action A.

A *global state* is a tuple of the form $(s_1, \ldots, s_I, v_1, \ldots, v_m)$ where s_i is the current local state of P_i and $v_1, \ldots, v_m$ is a list giving the current values of $x_1, \ldots, x_m$, respectively. A guard B_i is a predicate on global states, and an action A_i is a parallel assignment statement that updates the shared variables.

We model parallelism as usual by the nondeterministic interleaving of the "atomic" transitions of the individual processes P_i. Hence, at each step of the computation, some process with an "enabled" arc is nondeterministically selected to be executed next. Let $s = (s_1, \ldots, s_i, \ldots, s_I, v_1, \ldots, v_m)$ be the current global state, and let P_i contain an arc from node s_i to s'_i labeled with $B_i \to A_i$ (we write this arc as the tuple $(s_i, B_i \to A_i, s'_i)$). If B_i holds in s, then a permissible next state is $s' = (s_1, \ldots, s'_i, \ldots, s_I, v'_1, \ldots, v'_m)$ where $v'_1, \ldots, v'_m$ are the new values for the shared variables resulting from action A. The *transition relation* R is the set of all such triples (s, i, s'). The arc from node s_i to s'_i is *enabled* in state s. A *computation path* is a sequence of states $s^0, s^1, \ldots, s^k, \ldots$ where $\forall k \geq 0, \exists i \in [1:I] : (s^k, i, s^{k+1}) \in R$,[2] i.e., each successive pair of states is related by R. If $s = (s_1, \ldots, s_i, \ldots, s_I, v_1, \ldots, v_m)$, then we define $s{\upharpoonright}i = s_i$ and $s{\upharpoonright}\mathcal{A}P_i = V_i(s{\upharpoonright}i)$.

Definition 1 (Global state transition diagram). *Given a concurrent program $P = P_1 \| \cdots \| P_I$ and a set S_0 of initial global states for P, the* global state transition diagram *generated by P is a structure $M = (S_0, S, R, V)$ given as follows: (1) R is the next-state relation defined above, (2) S is the smallest set of global states satisfying (2.1) $S_0 \subseteq S$ and (2.2) if $\exists s \in S, i \in [1:I] : (s, i, t) \in R$ then $t \in S$, and (3) V is given by $V(s) = V_1(s_1) \cup \cdots \cup V_I(s_I)$, that is, a global state inherits its truth-assignments to atomic propositions from its constituent local states.*

[2] $[1:I]$ is the set of natural numbers from 1 to I, inclusive.

2.2 The Specification Language CTL

Our specification language is the propositional branching time temporal logic CTL [10]. CTL formulae are built up from the atomic propositions in $\mathcal{AP}$, $\neg$, $\wedge$, and the *temporal modalities* $\mathsf{EX}_i f$, $\mathsf{A}[f\mathsf{U}g]$, and $\mathsf{E}[f\mathsf{U}g]$ (f, g are sub-formulae).

Formally, we define the semantics of CTL formulae with respect to structures of the same type as global state transition diagrams, i.e., a structure $M = (S_0, S, R, V)$ consisting of a countable set S of global states, a set $S_0 \subseteq S$ of initial states, a relation $R \subseteq S \times [1:I] \times S$, giving the transitions, and a mapping $V : S \mapsto 2^{\mathcal{AP}}$ which labels each state s with a set $V(s) \subseteq \mathcal{AP}$ of atomic propositions true in s. If $s = (s_1, \ldots, s_i, \ldots, s_I, v_1, \ldots, v_m)$, then $V(s) \stackrel{\mathrm{df}}{=} V_1(s_1) \cup \cdots \cup V_I(s_I)$, where $V_i(s_i) \subseteq \mathcal{AP}_i$ gives the atomic propositions that hold in s_i. We require that R be total, i.e., that $\forall s \in S, \exists i, s' : (s, i, s') \in R$.

A *fullpath* is an infinite sequence of states $(s^0, s^1, \ldots, s^k, \ldots)$ such that $\forall k \geq 0, \exists i \in [1:I] : (s^j, i, s^{j+1}) \in R$, i.e., an infinite computation path. $M, s \models f$ means that f is true at state s in structure M. We define $\models$ inductively:

$$
\begin{array}{lll}
M, s \models p & \text{iff} & p \in V(s) \\
M, s \models \neg f & \text{iff} & \text{not}(M, s \models f) \\
M, s \models f \wedge g & \text{iff} & M, s \models f \text{ and } M, s \models g \\
M, s \models \mathsf{EX}_i f & \text{iff} & \text{for some state } t,\ (s, i, t) \in R \text{ and } M, t \models f, \\
M, s \models \mathsf{A}[f\mathsf{U}g] & \text{iff} & \text{for all fullpaths } (s, s^1, s^2, \ldots) \text{ in } M, \\
 & & \qquad \exists k \geq 0[M, s^k \models g \wedge (\forall \ell : 0 \leq \ell < k \Rightarrow M, s^\ell \models f)] \\
M, s \models \mathsf{E}[f\mathsf{U}g] & \text{iff} & \text{for some fullpath } (s, s^1, s^2, \ldots) \text{ in } M, \\
 & & \qquad \exists k \geq 0[M, s^k \models g \wedge (\forall \ell : 0 \leq \ell < k \Rightarrow M, s^\ell \models f)]
\end{array}
$$

Thus X indicates "nexttime" and U indicates "until": $[f\mathsf{U}g]$ means that g eventually holds, and f holds up to that point. E, A quantify existentially, universally (respectively), over the fullpaths starting from a state. A formula f is *satisfiable* if and only if there exists a structure M and state s of M such that $M, s \models f$. Such an M is a *model* of f. $M, U \models f$ abbreviates $\forall s \in U : M, s \models f$, 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 $(f \Rightarrow g) \wedge (g \Rightarrow f)$, $\mathsf{A}[f\mathsf{U_w}g]$ for $\neg\mathsf{E}[\neg g\mathsf{U}(\neg f \wedge \neg g)]$, $\mathsf{AF}f$ for $\mathsf{A}[\mathit{true}\mathsf{U}f]$, $\mathsf{AG}f$ for $\neg\mathsf{EF}\neg f$, $\mathsf{AX}_i f$ for $\neg\mathsf{EX}_i \neg f$, $\mathsf{EX}f$ for $\mathsf{EX}_1 f \vee \cdots \vee \mathsf{EX}_I f$, and $\mathsf{AX}f$ for $\mathsf{AX}_1 f \wedge \cdots \wedge \mathsf{AX}_I f$.

A formula of the form $\mathsf{A}[f\mathsf{U}g]$ or $\mathsf{E}[f\mathsf{U}g]$ is an *eventuality* formula. 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) fullpath starting at s, there exists a finite prefix of the fullpath in M whose last state satisfies g and all of whose other states satisfy f.

We annotate transitions in a structure with the index i of the process P_i executing the transition, and the assignment statement A (if any) that P_i executes, e.g., $s \xrightarrow{i,A} t$.

2.3 Example Specifications: mutual exclusion and readers-writers

The CTL specification of the two process mutual exclusion problem is the conjunction of the following ($i, j \in \{1, 2\}, i \neq j$):

$N_1 \wedge N_2$: Both processes are initially in their Noncritical region

$\mathsf{AG}(N_i \Rightarrow (\mathsf{AX}_i T_i \wedge \mathsf{EX}_i T_i))$: Any move P_i makes from its Noncritical region N_i is into its Trying region T_i and such a move is always possible.

$\mathsf{AG}(T_i \Rightarrow \mathsf{AX}_i C_i)$: Any move P_i makes from its Trying region T_i is into its Critical region C_i.

$\mathsf{AG}(C_i \Rightarrow (\mathsf{AX}_i N_i \wedge \mathsf{EX}_i N_i))$: Any move P_i makes from its Critical region C_i is into its Noncritical region N_i and such a move is always possible.

$\mathsf{AG}(N_i \equiv \neg(T_i \vee C_i)) \wedge \mathsf{AG}(T_i \equiv \neg(N_i \vee C_i)) \wedge \mathsf{AG}(C_i \equiv \neg(N_i \vee T_i))$: P_i is always in one of N_i, T_i, or C_i.

$\mathsf{AG}(N_i \Rightarrow \mathsf{AX}_j N_i) \wedge \mathsf{AG}(T_i \Rightarrow \mathsf{AX}_j T_i) \wedge \mathsf{AG}(C_i \Rightarrow \mathsf{AX}_j C_i)$: A transition by P_i cannot cause a transition by P_j (interleaving model of concurrency).

$\mathsf{AG}(T_i \Rightarrow \mathsf{AF} C_i)$: P_i does not starve.

$\mathsf{AG}(\neg(C_1 \wedge C_2))$: P_1, P_2 do not access their critical regions simultaneously.

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

To obtain the specification for readers-writers [6], we replace $\mathsf{AG}(T_i \Rightarrow \mathsf{AF}C_i)$ by the conjunction of the following, where P_1 is the reader and P_2 is the writer:

$\mathsf{AG}(T_1 \Rightarrow \mathsf{AF}(C_1 \vee \neg N_2))$: absence of starvation for reader provided writer does not request access

$\mathsf{AG}(T_2 \Rightarrow \mathsf{AF}C_2)$: absence of starvation for writer

$\mathsf{AG}((T_1 \wedge T_2) \Rightarrow \mathsf{A}[T_1 \mathsf{U} C_2])$: priority of writer over reader for access to Critical region

3 Overview of the CTL Decision Procedure

CTL is decidable: given a CTL formula f_0 there exists a decision procedure [10] that determines, in $O(2^{|f_0|})$ deterministic time, whether f_0 is satisfiable or not. The CTL decision procedure first constructs a particular kind of AND/OR graph (a tableau) T_0 for f_0. We use $c, c', \ldots$ to denote AND-nodes, $d, d', \ldots$ to denote OR-nodes, and $e, e', \ldots$ to denote nodes of either type. Each node e is labeled with a set of formulae $L(e)$, each of which is either a subformula of f_0, or a subformula of f_0 preceded by AX or EX. No two AND-nodes (OR-nodes) have the same label.

The CTL decision procedure constructs T_0 by starting with a single "root" OR-node d_0 labeled with $\{f_0\}$, and repeatedly constructing successors of "frontier" nodes until there is no change. The set of AND-node successors $\mathit{Blocks}(d)$ of an OR-node d is determined by expanding d into a tree as follows. A CTL formula is *elementary* iff it is an atomic proposition, the negation of an atomic proposition, or has either AX_i or EX_i as its main connective. We classify a nonelementary formula as either a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$ according to the fixpoint characterization of the main connective, e.g., $\mathsf{AG}g \equiv g \wedge \mathsf{AXAG}g$, so $\alpha_1 = g$, $\alpha_2 = \mathsf{AXAG}g$, and $\mathsf{AG}g$ is a α formula, and $\mathsf{AF}g \equiv g \vee \mathsf{AXAF}g$, so $\beta_1 = g$, $\beta_2 = \mathsf{AXAF}g$, and $\mathsf{AF}g$ is a β formula. Suppose e is a leaf in the tree constructed so far, and $f \in L(e)$. If $f \equiv \alpha_1 \wedge \alpha_2$, then add a single son to e with label $L(e) - \{f\} \cup \{\alpha_1, \alpha_2\}$. If $f \equiv \beta_1 \vee \beta_2$, then add two sons to e with labels $L(e) - \{f\} \cup \{\beta_1\}$, $L(e) - \{f\} \cup \{\beta_2\}$. This tree construction terminates when all leaves contain only elementary formulae in their labels. This must happen, since each expansion removes one nonelementary formula and replaces it with one or two smaller formulae. Upon termination, let $\mathit{Blocks}(d)$ contain one AND-node c for each leaf node, whose label $L(c)$ is the union of all node labels along the path from the corresponding leaf back to the root d of the tree. The nodes in $\mathit{Blocks}(d)$ embody all the different ways in which the (conjunction of the) formulae in $L(d)$ can be satisfied: $L(d)$ is satisfiable iff $L(c)$ is satisfiable for at least one $c \in \mathit{Blocks}(d)$. In the final tableau, an OR-node must have at least one AND-node successor present.

The set $\mathit{Tiles}(c)$ of OR-node successors of an AND-node c is $\bigcup_{i \in [1:I]} \mathit{Tiles}_i(c)$, where $\mathit{Tiles}_i(c)$ is the set of OR-node successors of c that are associated with P_i. Suppose that c is labeled with n formulae of the form $\mathsf{AX}_i g$, namely $\mathsf{AX}_i g_1, \ldots,$ $\mathsf{AX}_i g_n$, and m formulae of the form $\mathsf{EX}_i h$, namely $\mathsf{EX}_i h_1, \ldots, \mathsf{EX}_i h_m$. Then $\mathit{Tiles}_i(c) \stackrel{\mathrm{df}}{=} \{d_i^1, \ldots, d_i^m\}$, where $L(d_i^j) = \{\mathsf{AX}_i g_1, \ldots, \mathsf{AX}_i g_n\} \cup \{\mathsf{EX}_i h_j\}$, for $j \in [1:m]$. Finally, the edge from c to every node in $\mathit{Tiles}_i(c)$ is labeled with the process index i, to indicate that this successor is associated with P_i. $\mathit{Tiles}(c)$ is exactly the set of successors required to satisfy all of the nexttime formulae in the label of c: $L(c)$ is satisfiable iff $L(d)$ is satisfiable for all $d \in \mathit{Tiles}(c)$, and $LP(c)$ is satisfiable, where $LP(c) = \{f \in L(c) \mid f \text{ is a proposition or its negation}\}$.

We continue generating successors of frontier nodes ("expanding" a node) until there are no more frontier nodes, i.e., every node in T_0 has at least one successor. If a node is ever created which has the same label as an already present node of the same type (i.e., AND or OR), then we merge the two nodes. Since the number of possible labels is finite ($O(2^{|f_0|})$), this process terminates.

The next step is to apply the *deletion rules* given in Figure 1 to T_0. Roughly speaking, these rules remove all nodes e whose label is propositionally inconsistent, or who do not have enough successors, or who are labeled with an eventuality formula which is not fulfilled. The presence of a suitable *full subdag* (path) rooted at e serves to certify the fulfillment of an eventuality $\mathsf{A}[g\mathsf{U}h]$ ($\mathsf{E}[g\mathsf{U}h]$) in $L(e)$. A full subdag D rooted at node e in T_0 is a directed acyclic subgraph of T_0 such that: (1) e is the unique node from which all other nodes in D are reachable, (2) for every AND-node c in D, if c has any sons in D, then every successor of c in T_0 is a son of c in D, and (3) for every OR-node d in D, there exists precisely one AND-node c in T_0 such that c is a son of d in D. We repeatedly apply the deletion rules until there is no change. Since each application removes one node, and T_0 is finite, this procedure must terminate. Upon termination, if the root of T_0 is has been removed, then f_0 is unsatisfiable. Otherwise f_0 is satisfiable, in which case let T^* be the tableau induced by the remaining nodes.

For each eventuality $\mathsf{A}[g\mathsf{U}h] \in L(c)$, let $\mathrm{DAG}[c, \mathsf{A}[g\mathsf{U}h]]$ be the directed acyclic graph that results from removing all the OR-nodes in a full subdag D rooted at c that fulfills $\mathsf{A}[g\mathsf{U}h]$, and for each eventuality $\mathsf{E}[g\mathsf{U}h] \in L(c)$, let $\mathrm{DAG}[c, \mathsf{E}[g\mathsf{U}h]]$ be the path that results from removing all the OR-nodes in a path starting from c that fulfills $\mathsf{E}[g\mathsf{U}h]$. In both cases we connect up the AND-nodes so that $c' \to c''$ in $\mathrm{DAG}[c, g]$ only if $c' \to d \to c''$ for some removed OR-node d. These DAG's exist by virtue of Figure 1.

For each AND-node c in T^*, we construct a "fragment" FRAG[c] by connecting up copies of the DAG's for the eventualities in $L(c)$, so that for $\mathsf{A}[g\mathsf{U}h] \in L(c)$, every infinite path from c encounters DAG[$c, \mathsf{A}[g\mathsf{U}h]$], and for $\mathsf{E}[g\mathsf{U}h] \in L(c)$, some infinite path from c has DAG[$c, \mathsf{E}[g\mathsf{U}h]$] as a prefix. Thus, all eventualities in $L(c)$ are fulfilled in FRAG[c]. We construct a model M for f_0 by connecting up copies of all the FRAG's so that every state (AND-node) c has at least one successor. This is done by identifying the root of one FRAG with a frontier node of another FRAG if they have the same label. The truth assignment V is given by $V(c) = L(c) \cap \mathcal{AP}$, where $\mathcal{AP}$ is the set of atomic propositions in *spec*. In M, every state satisfies all the formulae in its label. From M, a correct concurrent program can be produced by projecting onto the individual processes, as given in Definition 2 below.

DeleteP Delete any propositionally inconsistent node.

DeleteOR Delete any OR-node all of whose successors are already deleted.

DeleteAND Delete any AND-node one of whose successors is already deleted.

DeleteAU Delete any node e such that $\mathsf{A}[g\mathsf{U}h] \in L(e)$ and there does not exist a full subdag rooted at e where $h \in L(c')$ for every frontier node c' and $g \in L(c'')$ for every interior AND-node c''.

DeleteEU Delete any node e such that $\mathsf{E}[g\mathsf{U}h] \in L(e)$ and there does not exist an AND-node c' reachable from e via a path π such that $h \in L(c')$ and for all AND-nodes c'' along π up to but not necessarily including c', $g \in L(c'')$.

Fig. 1. The deletion rules for the CTL decision procedure.

4 Refinability of specifications

4.1 Implementing the guards: temporary stability

Suppose that in a program $P = P_1 \| \cdots \| P_I$, a guard B_i of an arc $a_i = (s_i, B_i \rightarrow A_i, t_i)$ of process P_i is *temporarily stable*, [12], that is, once B_i holds, it continues to hold until P_i executes some transition, not necessarily a transition corresponding to the execution of a_i. In this case, P_i can test for the truth of B_i by repeatedly reading the individual variables referenced in B_i. More formally, let $(s_i, B_i \rightarrow A_i, t_i)$ be an arc of P_i, and let $M = (S_0, S, R, V)$ be the global state transition diagram of P given by Definition 1. We require

$$M, S_0 \models \mathsf{AG}(\ (\{\!\!\{s_i\}\!\!\} \wedge B_i) \Rightarrow \mathsf{A}[B_i \,\mathsf{U_w}\, \neg\{\!\!\{s_i\}\!\!\}]\ ). \qquad \text{(GSTAB)}$$

where $\{\!\!\{s_i\}\!\!\} =$ "$(\bigwedge_{Q \in \mathcal{AP}_i \cap V_i(s_i)} Q) \ \wedge\ (\bigwedge_{Q \in \mathcal{AP}_i - V_i(s_i)} \neg Q)$". $\{\!\!\{s_i\}\!\!\}$ characterizes s_i in that $s_i \models \{\!\!\{s_i\}\!\!\}$, and $s'_i \not\models \{\!\!\{s_i\}\!\!\}$ for all local states s'_i such that $s'_i \neq s_i$, i.e., it converts a local state into a propositional formula. GSTAB requires that once P_i is in state s_i and guard B_i holds, then B_i continues to hold until P_i leaves s_i, if ever. Note the use of the *weak until* $\mathsf{U_w}$: $[B_i \,\mathsf{U_w}\, \neg\{\!\!\{s_i\}\!\!\}]$ means that B_i

holds until $\neg\{\!\{s_i\}\!\}$ becomes true (i.e., P_i leaves s_i), or, B_i holds forever if $\neg\{\!\{s_i\}\!\}$ never becomes true. Thus, P_i can check B_i by reading the atomic propositions and shared variables in B_i sequentially, i.e., in a non-atomic manner. If P_i ever observes that B_i holds, then P_i can subsequently execute a_i.

We say that "M satisfies GSTAB" if and only if GSTAB holds for every arc $(s_i, B_i \to A_i, t_i)$ of every process P_i of P.

Given a CTL formula *spec*, we wish to answer the following question: does there exist a program P which both satisfies *spec* and whose guards are temporarily stable? More technically, does there exist a program P with global state transition diagram $M = (S_0, S, R, V)$ such that $M, S_0 \models spec$, and M satisfies GSTAB? Since the tableau T^* for *spec* that is generated by the CTL decision procedure encodes every possible model of *spec*, we can answer this question by analyzing T^*. Figure 2 presents an algorithm which performs this analysis.

To explain the operation of the algorithm, we first discuss how we extract a program from a structure M that conforms to the interleaving model, i.e., only transitions by P_i change atomic propositions in $\mathcal{A}P_i$. A *P_i-family* [3] F in $M = (S_0, S, R, V)$ is a maximal subset of R such that (1) all members of 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 F: $s{\upharpoonright}i = s'{\upharpoonright}i$ and $t{\upharpoonright}i = t'{\upharpoonright}i$. If $s \xrightarrow{i,A} t \in F$, then let $F.start$, $F.finish$, $F.assig$, $F.label$ denote $s{\upharpoonright}i$, $t{\upharpoonright}i$, 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 $F.guard$ denote $\bigvee_{T \in F} \{\!\{(T.begin){\downharpoonleft}i\}\!\}$, where $s{\downharpoonleft}i$ is s with its P_i-component removed, and $\{\!\{s{\downharpoonleft}i\}\!\} =$ "$(\bigwedge_{Q \in (\mathcal{A}P - \mathcal{A}P_i) \cap V(s)} Q) \;\; \wedge \;\; (\bigwedge_{Q \in (\mathcal{A}P - \mathcal{A}P_i) - V(s)} \neg Q) \;\; \wedge \;\; (\bigwedge_x x = s(x))$", where x ranges over the shared variables. $\{\!\{s{\downharpoonleft}i\}\!\}$ converts global state s into an "equivalent" propositional formula, with the omission of the component $s{\upharpoonright}i$.

Definition 2 (Program Extraction). *Let $M = (S_0, S, R, V)$ be a structure that conforms to the interleaving model. Then the program $P = P_1 \| \cdots \| P_I$ extracted from M is as follows. Process P_i contains arc $(s_i, B_i \to A_i, t_i)$ if and only if:*

there exists a P_i-family F in M such that

$$F.start = s_i,\ F.finish = t_i,\ F.assig = A_i,\ F.guard = B_i.$$

The truth assignment V_i is given by $V_i(s_i) = V(s) \cap \mathcal{A}P_i$ where $s \in S$ is such that $s{\upharpoonright}i = s_i$.

The key idea is this: for the guard B_i to be temporarily stable, we need that, once a global state s is entered which has an outgoing transition belonging to F, i.e., $s{\upharpoonright}i = s_i$ and $\exists t : s \xrightarrow{i,A} t \wedge t{\upharpoonright}i = t_i$, then every transition by some process other than P_i must lead to a state which also has an outgoing transition belonging to F, i.e., to a state u such that $u{\upharpoonright}i = s_i$ and $\exists v : u \xrightarrow{i,A} v \wedge v{\upharpoonright}i = t_i$.

Consider AND-node c which has an outgoing AND-OR transition $t = c \xrightarrow{i} d$. If c is present as a state in the final extracted model M, then there will be an outgoing transition from c (in M) corresponding to the AND-OR transition t. This transition is a member of a family F. To check that M satisfies the above condition, we check that T^*, from which M is extracted, satisfies an analogous condition, applied to the AND-nodes of T^*, which become states in M. The algorithm of Figure 2 performs this check as follows. First invoke the CTL decision procedure on *spec*, halting if *spec* is unsatisfiable. If not, then analyze the tableau T^* as follows. For every AND-node in T^*, compute the set C of all AND-nodes reachable from c by paths not labeled with index i, i.e., corresponding to executions by processes other than P_i. Then, check every AND-node c' in C to ensure that it has an outgoing AND-OR transition $c' \xrightarrow{i} d'$ in T^* such that $d' \upharpoonright \mathcal{A}P_i = d \upharpoonright \mathcal{A}P_i$, i.e., an AND-OR transition that will generate, in the extracted model M, a transition in family F. If not, then c' causes a violation of GSTAB, and must be made unreachable from c by deleting all of the OR-AND transitions from OR-nodes in C to c'. If all such necessary deletions can be made without causing the root node to be deleted, according to the deletion rules of Figure 1, then a model M can be extracted from the resulting tableau, using the same method as in the CTL decision procedure, and M will satisfy GSTAB.

---

1. Apply the CTL decision procedure to *spec*. If the root of T_0 is deleted, then
   output "there exists no program satisfying *spec*" and halt.
   Otherwise, let T^* be the resulting tableau.
2. **for** every process index i, and every AND-OR transition $t = c \xrightarrow{i} d$ in T^*:
   - $C := \{e \mid e$ is reachable from c by a path not containing process index $i\}$;
   - **forall** AND-nodes $c' \in C$ in increasing distance from c **do**
     - **if** there exists an AND-OR transition $c' \xrightarrow{i} d'$ in T^* such that $d' \upharpoonright \mathcal{A}P_i = d \upharpoonright \mathcal{A}P_i$ **then**
       - mark c' as "satisfying with respect to t"
     - **else**
       - delete all the OR-AND transitions from OR-nodes in C to c';
       - recompute C to account for the deletion of the OR-AND transitions
     - **endif**
   - **endfor**;

/* call the resulting tableau T_s */

3. Apply the deletion rules of Figure 1 to T_s;
4. **if** the root node of T_s is undeleted **then** /* positive decision */
   - let T be the subgraph of T_s induced by the remaining undeleted nodes;
   - extract M from T using the same method as in the CTL decision procedure

   else /* negative decision */
   - output "there exists no program satisfying the specification whose guards are temporarily stable"

   endif

Fig. 2. The Test for Specifications that allow Temporarily Stable guards

Shared Variables The algorithm of Figure 2 does not take shared variables into account. We introduce shared variables to distinguish between global states which have different labels, but which assign the same values to all atomic propositions [10]. This is necessary, since only atomic propositions are implemented in the synthesized program, whereas the labels which distinguish different states in the tableau consist of not only atomic propositions, but CTL formulae in general. Thus, if propositionally identical but globally different states are not distinguished, the effect would be to "merge" such states, which could lead to violation of liveness, e.g., if the $[T_1\ T_2]$ states c_6 and c_7 in Figure 3 are merged in this way, then the liveness specification $\mathsf{AG}(T_i \Rightarrow \mathsf{AF}C_i)$, $i \in \{1,2\}$, is violated. So, in Figure 3, we introduce a shared variable x which has value 1 in c_6 and value 2 in c_7. This requires adding an assignment $x := 1$ to all transitions entering c_6, and an assignment $x := 2$ to all transitions entering c_7. Whilst x will appear in the guards of the synthesized program, the temporary stability of these guards is dependent solely on the existence of the appropriate AND-OR transitions $c' \xrightarrow{i} d'$ as determined by the algorithm of Figure 2. The *subsequent* introduction of a shared variable does not change this, provided however, that the assignment to the shared variable is performed along all transitions of P_i which belong to the same transition family.

Theorem 1. *Let spec be a CTL formula, and suppose that the algorithm of Figure 2 produces a model M when applied to spec. Then, M satisfies GSTAB.*

4.2 Implementing the multiple assignments: lock-free multi-object operations

Execution of an arc $(s_i, B_i \to A_i, t_i)$ involves both changing the atomic propositions in $\mathcal{A}P_i$ which are true from those in $V_i(s_i)$ to those in $V_i(t_i)$ (all other atomic propositions remaining unchanged) and updating the shared variables according to the parallel assignment A_i, which has the form $x, y, \ldots := v, w, \ldots$ where $x, y, \ldots$ is a list of shared variables, and $v, w, \ldots$ is a list of constants.

We implement this as follows. First, we consolidate all the atomic propositions of each P_i into a single variable L_i, whose value in local state s_i is $V_i(s_i)$: $s_i(L_i) = V_i(s_i)$, i.e., L_i is the set of atomic propositions in $\mathcal{A}P_i$ that are true in s_i. In practice, L_i could be encoded efficiently as a bit string. Thus, in executing the arc $(s_i, B_i \to A_i, t_i)$, we update the value of L_i from $V_i(s_i)$ to $V_i(t_i)$. We now have a multiple assignment of the form $L_i, x, y, \ldots := V_i(t_i), v, w, \ldots$. To implement this multiple assignment, we use any lock-free method for implementing multiple-object operations atomically [1, 11, 16, 19]. We do not need the more expensive wait-free implementations, because we only need to correctly implement the transitions in the model M, and, a lock-free implementation suffices for this. Liveness properties are still satisfied, since M satisfies liveness properties under *nondeterministic* scheduling, i.e., no matter which transition is next selected for execution. In particular, no form of fairness is needed.

4.3 Implementation in hardware-available primitives

Let M be a model for *spec* resulting from the algorithm of Figure 2, and let P be a program extracted from M according to Definition 2. Let $M_P = (S_0, S, R, V)$ be the global state transition diagram of P given by Definition 1. Then, $M_P, S_0 \models$ *spec* by the soundness of the CTL decision procedure [10]. Also, M_P satisfies GSTAB, since we can show that M_P and M are strongly bisimilar [5]. Let $(s_i, B_i \to A_i, t_i)$ be an arc of P_i in program P, where A_i is $x, y, \ldots := v, w, \ldots$. We implement this arc as follows:

1. **while** the guard B_i is not observed to be true
   read sequentially all the atomic propositions and shared variables in B_i;
   evaluate B_i
   endwhile;
2. Invoke a lock-free multiple object operation to implement the multiple assignment $L_i, x, y, \ldots := V_i(t_i), v, w, \ldots$.

We show that this implementation of P is correct by establishing a *stuttering bisimulation* [5] between M_P and the global-state transition diagram M_{imp} of the implementation, which is formally defined along the lines of Definition 1. See [4] for examples of such definitions for low-atomicity implementations. A state s of M and a state u of M_{imp} are related by stuttering bisimulation iff they assign the same values to all atomic propositions and shared variables. Since states related by stuttering bisimulation satisfy the same formulae of CTL − X (CTL without the EX_i, AX_i modalities) this is sufficient to establish typical safety and liveness properties. Also, if a conjunct of *spec* has the forms $\mathsf{AG}(p_i \Rightarrow \mathsf{AX}_i q_i)$, $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$, then $\mathsf{AG}(p_i \Rightarrow \mathsf{AX}_i(p_i \vee q_i))$, $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i(p_i \vee q_i))$, respectively, is satisfied by the implementation, where p_i, q_i specify local states of P_i. We defer details of this to the full paper.

Theorem 2. *Let spec be a CTL formula, and suppose that the algorithm of Figure 2 produces a model M of spec. Let P be the program extracted from M by Definition 2, let M_{imp} be the global state transition diagram of the implementation of P given above, and let S^0_{imp} be the set of initial states of M_{imp}. Let f be a conjunct of spec which contains no EX_i or AX_i modality. Then $M_{imp}, S^0_{imp} \models f$.*

5 Examples: mutual exclusion and readers-writers

We now apply the above test to the mtual exclusion and readers-writers specifications. Figure 3 shows the tableau produced by the CTL decision procedure for the mutual exclusion specification given in Section 2.3. The OR-nodes are named d_k, and the AND-nodes are named c_k. These names are not part of the decision procedure, and are provided only to facilitate the discussion. The initial OR-node is d_0. Upon applying the algorithm of Figure 2 to the tableau of Figure 3, we find that the tableau passes the test. Consider, for exampe, the transition $t = c_1 \xrightarrow{1} d_5$, in which P_1 moves from T_1 to C_1, and the application of the test to t. The set of nodes reachable from c_1 by a path not containing process index 1 is $\{d_6, c_6, c_7, d_{11}, c_{10}, d_1, c_2\}$. AND-node c_6 is marked as "satisfying w.r.t. t", since c_6 has an OR-node successor d_{10} which reflects the same transition by P_1, namely from T_1 to C_1. AND-node c_7, on the other hand, fails the test, since it does not have a suitable OR-node successor. Hence, the OR-AND transition

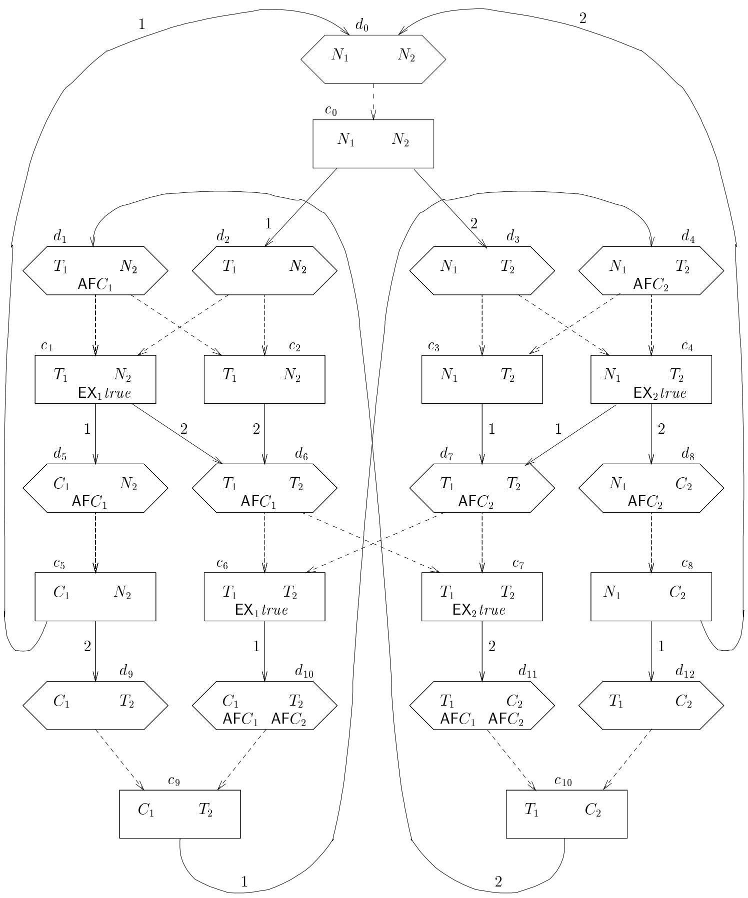

Fig. 3. Tableau for the mutual exclusion specification

from d_6 to c_7 is deleted. This causes the remaining nodes to become unreachable from c_1 by paths not containing process index 1, and so we are done. The tableau as a whole remains viable, since d_6 still has a single successor, c_6. For reasons of symmetry, d_7 will be left with sole successor c_7 when the test is applied to transition $c_4 \xrightarrow{2} d_8$. Thus, the root is not deleted, and the synchronization skeletons shown in Figure 4 can be extracted from the tableau.

Figure 5 shows the tableau produced by the CTL decision procedure for the readers-writers specification given in Section 2.3. The initial OR-node is d_0. Consider the transition $t = c_1 \xrightarrow{1} d_5$, in which P_1 moves from T_1 to C_1, and the application of the test to t. The set of nodes reachable from c_1 by a path not containing process index 1 is $\{d_6, c_7, d_{11}, c_{10}, d_1, c_2\}$. The AND-node c_7 fails the test, since it does not have a suitable OR-node successor. Hence, the OR-AND transition from d_6 to c_7 is deleted. This now leaves d_6 without a successor.

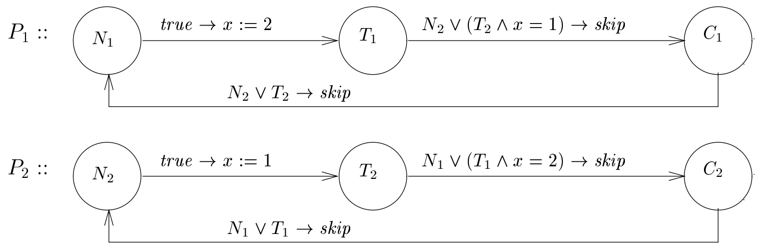

Fig. 4. Synchronization skeleton program for the mutual exclusion specification

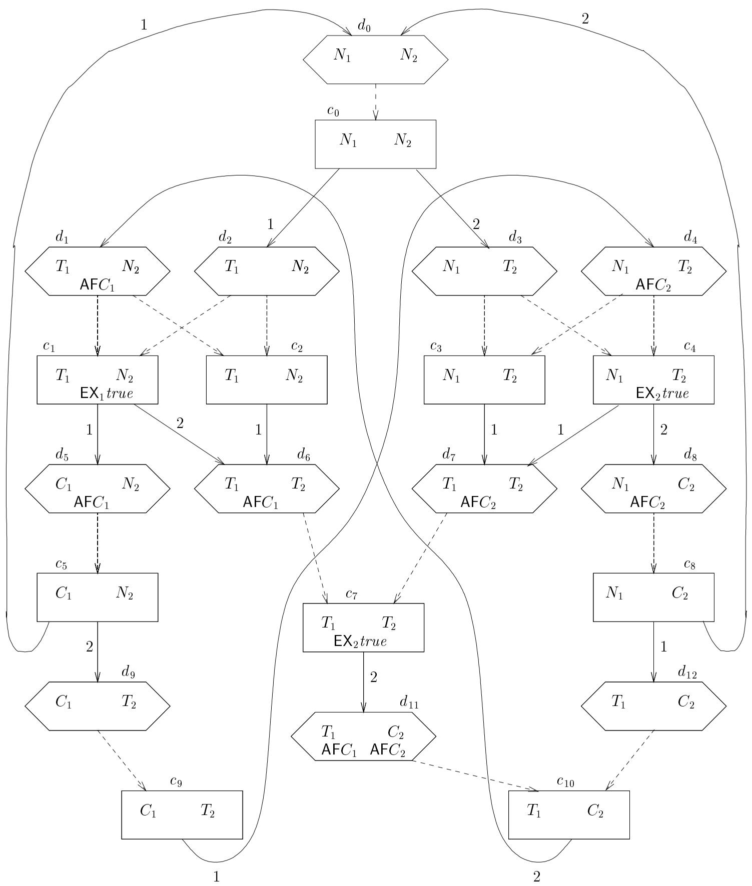

Fig. 5. Tableau for the readers-writers specification

Hence, when the deletion rules of Figure 1 are applied, d_6 is deleted. This, in turn results in the deletion of c_1 and c_2, since they are AND-nodes, and so require all successors to be undeleted. This results in the deletion of d_1 and d_2 since they are left without successors. The deletion of d_2 causes the deletion of AND-node c_0, and this causes the deletion of the root node d_0, since c_0 is the only successor of d_0. Thus the tableau is not viable, and we conclude that there exists no concurrent program which satisfies the readers-writers specification and which has temporarily-stable guards.

Intuitively, we see that the readers-writers specification imposes a "flickering" guard on the reader, since it allows the writer to always preempt the reader's ability to enter the critical section: when the writer is in N_2 and the reader in T_1, the reader is enabled to enter C_1, but the writer can autonomously preempt this enablement by entering T_2. This is inherent in the writer priority requirement of the specification.

6 Conclusions and Further Work

We presented a method for deciding whether a specification can be implemented by a concurrent program which has the property of being "easily" refined to a low-grain atomicity program that uses primitives available in hardware. The refinement process is automatic, and the final program does not resort to inefficient strategies such as using a central module which controls everything. In practice, our method can be used iteratively. If the procedure of Figure 2 outputs "no" for a given specification *spec*, then every program which satisfies *spec* must contain "flickering" guards, which can transit from true to false before the arc that they label is executed. Detecting the truth of such guards is difficult: it requires high atomicity operations, or inefficient strategies such as blocking or centralization. In this case, the best course of action may be to modify the specification and reapply the method. Extending the method to give advice on modifying the specification so that it passes the test of Figure 2 is a topic of future work.

Our test can be viewed as a *design rule*: specifications which fail it are in some sense bad specifications, as they necessitate inefficient programs. Our result therefore contributes to software engineering, as it provides a criterion for judging the quality of a specification. More generally, our work suggests a notion of *implementation complexity* for specifications: can we define a complexity measure on specifications which indicates the "difficulty" of implementing a concurrent program P that satisfies the specification. This "difficulty" may take several attributes into account: the amount of blocking and centralization in P, the length of the proof that P satisfies the specification, etc. We will examine this issue further in future work.

References

1. James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Symposium on Principles of Distributed Computing*, 1995.

2. A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. In *Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 156–169, Berlin, 1994. Springer-Verlag.
3. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems for an atomic read/atomic write model of computation (extended abstract). In *Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 111–120, Philadelphia, Pennsylvania, May 1996. ACM Press.
4. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems for an atomic read/write model of computation. *ACM Trans. Program. Lang. Syst.*, 23(2):187–242, Mar. 2001. Extended abstract appears in ACM Symposium on Principles of Distributed Computing (PODC) 1996.
5. M.C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
6. P.J. Courtois, H. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667 – 668, 1971.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.
8. D.L. Dill and H. Wong-Toi. Synthesizing processes and schedulers from temporal specifications. In *International Conference on Computer-Aided Verification*, number 531 in LNCS, pages 272–281. Springer-Verlag, 1990.
9. E. A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, Mass., 1990.
10. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2:241 – 266, 1982.
11. Timothy L Harris, Keir Fraser, and Ian Pratt. A practical multi-word compare-and-swap operation. In *IEEE Symposium on Distributed Computing*, 2002.
12. S. Katz. Temporary stability in parallel programs. Tech. Rep., Computer Science Dept., Technion, Haifa, Israel, 1986.
13. O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc. 11th Int. Conf. on Concurrency Theory (CONCUR)*, Springer LNCS volume 1877, pages 92–107.
14. O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *2nd International Conference on Temporal Logic*, pages 91–106, Manchester, July 1997. Kluwer Academic Publishers.
15. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, Jan. 1984. Also appears in Proceedings of the Workshop on Logics of Programs, Yorktown-Heights, N.Y., Springer-Verlag Lecture Notes in Computer Science (1981).
16. M. Moir. Transparent support for wait-free transactions. In *Workshop on Distributed Algorithms*, 1997.
17. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 179–190, New York, 1989. ACM.
18. A. Pnueli and R. Rosner. On the synthesis of asynchronous reactive modules. In *Proceedings of the 16th ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671, Berlin, 1989. Springer-Verlag.
19. N. Shavit and D. Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, Ontario, Canada, 1995.