Reasoning about Systems with Many Processes

STEVEN M. GERMAN

GTE Laboratories, Inc., Waltham, Massachusetts

AND

A. PRASAD SISTLA

University of Illinois at Chicago, Chicago, Illinois

Abstract. Methods are given for automatically verifying temporal properties of concurrent systems containing an arbitrary number of finite-state processes that communicate using CCS actions. Two models of systems are considered. Systems in the first model consist of a unique *control* process and an arbitrary number of *user* processes with identical definitions. For this model, a decision procedure to check whether all the executions of a process satisfy a given specification is presented. This algorithm runs in time double exponential in the sizes of the control and the user process definitions. It is also proven that it is decidable whether all the fair executions of a process satisfy a given specification. The second model is a special case of the first. In this model, all the processes have identical definitions. For this model, an efficient decision procedure is presented that checks if every execution of a process satisfies a given temporal logic specification. This algorithm runs in time polynomial in the size of the process definition. It is shown how to verify certain global properties such as mutual exclusion and absence of deadlocks. Finally, it is shown how these decision procedures can be used to reason about certain systems with a communication network.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*concurrency*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program verification*

General Terms: Algorithms, Performance, Theory, Verification

1. *Introduction*

Automatic verification of finite-state concurrent systems has recently been an active area of research. Many different algorithms for checking if a finite-state concurrent system meets a specification given in a Temporal Logic have been proposed in the literature [2, 5, 16, 29, 33]. Some of these algorithms have been implemented and have been successfully used to automatically verify systems such as concurrent programs and hardware designs. All the previously mentioned algorithms assume that the global behavior of the concurrent system can be modeled by a finite-state graph. They first construct the global-state graph and verify that it satisfies the given specification. However, there are many concurrent and distributed systems that are designed to operate with an

Authors' addresses: S. M. German, GTE Laboratories, Inc., 40 Sylvan Road, Waltham, Mass., 02154; A. P. Sistla, Department of Electrical Equipment and Computer Sciences, University of Illinois at Chicago, Chicago, IL, 60680.

arbitrary number of processes. Clearly, the previous techniques cannot be applied to such systems. The problem of extending the previous techniques to systems of many identical processes has been addressed in [4], where it is shown that under certain conditions and for some properties, the correctness of a system with many processes can be inferred from the correctness of the system with two processes. This approach is only partly automatic. In this paper, we present, for the first time, fully automatic methods for verifying certain properties of concurrent systems with an arbitrary number of communicating processes. Our methods are complete for verifying a well-defined class of properties in certain natural models of computation.

Our specification language is PTL, which is the standard *Propositional Linear Temporal Logic*. We assume that the concurrent system has an arbitrary number of processes. The processes communicate through synchronous actions in the style of Milner's Calculus of Communicating Systems (CCS) [20]. A computation step of a system consists of either one process taking an internal step or any two processes that have enabled transitions on complementary actions synchronizing and taking a step together. (The semantics of processes is defined formally in Section 2.) Given such a system, we investigate the problem of checking if all the executions of a process satisfy a specification given in PTL (or given by a finite-state automaton on infinite strings). We consider two different models.

Systems in the first model consist of a unique *control* process and arbitrary number of *user* processes with identical definitions. Some resource allocation methods and network protocols can be defined in this model at a certain level of abstraction. For this model, we present a decision procedure that checks if the executions of the processes satisfy a given specification. The algorithm first constructs a *Vector Addition System with States* (VASS) that captures the behavior of the concurrent system and then checks if the VASS has an infinite path that contains a final state occurring infinitely often. This algorithm runs in time double exponential in the size of the definitions of the processes and the size of the temporal specification.

In concurrent systems, many times we are interested in verifying that the correctness specification is satisfied by all the *fair* computations. This is especially the case for *liveness* properties. For this reason, we consider the problem of checking if all the executions of a process in fair computations of a system satisfy a given specification. We present a decision procedure for this problem for systems in the above model of computation. The decision procedure uses the decision procedure for the *reachability problem* of vector addition systems. We also show that this problem is as hard as the reachability problem for vector addition systems. This result indicates that any decision procedure for this problem will probably have much higher complexity than the algorithm mentioned above for the case when fairness is not considered.

In the second model, we assume that the system contains an arbitrary finite number of processes with identical definitions. Clearly, we can consider this model as a special case of the previous model and use the decision procedures of the previous model. However, we present a more efficient decision procedure for this model for checking if all the executions of a process satisfy a given specification. This decision procedure is based on the idea that the set of executions of a single process in a computation of the system is exactly the set of strings accepted by a certain finite-state automaton on infinite strings. The

size of this automaton is linear in the size of the process definition. We present an algorithm to construct this automaton in time polynomial in the size of the process. For the given PTL specification f, we construct the automaton $A_{\neg f}$ that accepts exactly the set of strings that satisfy $\neg f$, and we check that the intersection of this and the automaton for the process is empty. The whole algorithm runs in time $O(p(n) \cdot 2^{|f|})$ where $p(n)$ is a polynomial in n, the size of a process. The same procedure can also be used for systems with a fixed number of classes, where each class contains an arbitrary number of processes that have identical definitions.

Although we consider systems of many processes, the logic we use, PTL, only specifies the properties of the executions of single processes in a system. However, some of the important properties, such as *mutual exclusion* and *absence of deadlock* are *global properties* of the computations. We show that there is a straightforward way to verify mutual exclusion by verifying a certain property of a single process in a modified system. We also consider the problem of checking for potential *deadlocks*, that is, checking whether there exists a number of processes n such that there is a computation in the system of size n that ends in a deadlock. We show that this problem is decidable, but is as hard as the reachability problem for vector addition systems. We also show that in our first model, we can handle systems of processes with communication ports. The communication ports allow the control process to communicate with one of the users and then carry out a sequence of further communications with that user.

Finally, we extend PTL by allowing quantifiers over processes. This is a very natural extension of PTL and allows us to specify a wider class of properties of concurrent systems with many processes. This logic is similar to *indexed CTL** (ICTL*) considered in [4] except that we use linear time logic instead of branching time logic. We show that the the problem of verifying that a system in the first model satisfies a given specification in the extended logic is undecidable.

The rest of this paper is organized as follows: Section 2 gives definitions and notation. Section 3 presents the algorithms for analyzing a system containing a unique control processes and many user processes. Section 4 presents the results for a system where all the processes have identical definitions. Section 5 shows how to reason about certain global properties and some systems with a communication network. Section 6 discusses the extended logic *indexed PTL* (IPTL). Finally, Section 7 discusses related work and contains our conclusions.

2. *Background and Definitions*

2.1. Temporal Logic and Automata. The specification language we use is PTL, which is Propositional Linear Temporal Logic. The language of PTL uses a finite set $\mathcal{P}$ of atomic propositions, the constant True, the connectives $\neg$, $\supset$, and the temporal modalities **X** (*next time*) and **U** (*until*). The set of PTL formulas is the smallest set satisfying the following closure condition: the constants and atomic propositions are formulas, and if f and g are formulas, then $\neg(f)$, $(f \supset g)$, $\mathbf{X}(f)$, and $(f\ \mathbf{U}\ g)$ are formulas. An *interpretation* is a pair (t, i) where $t = (t_0, t_1, \ldots)$ is an ω-sequence of subsets of $\mathcal{P}$ and i is a nonnegative integer. Intuitively, t_i specifies the set of atomic propositions that are true at time i. We write $(t, i) \models f$ to denote that the interpretation (t, i)

satisfies the formula f. We say that t satisfies f if $(t, 0)$ satisfies f. The relation $\models$ is defined by induction on the structure of formulas:

$(t, i) \models \text{True};$

$(t, i) \models P$, where P is an atomic proposition, iff $P \in t_i$;

$(t, i) \models \neg f$ iff it is not the case that $(t, i) \models f$;

$(t, i) \models (f \supset g)$ iff either $(t, i) \models g$ or $(t, i) \models \neg f$;

$(t, i) \models \mathbf{X}\, f$ iff $(t, i + 1) \models f$.

$(t, i) \models (f\ \mathbf{U}\ g)$ iff there exists $k \geq i$ such that $(t, k) \models g$, and for all j, $i \leq j < k$, $(t, j) \models f$.

We also use the unary temporal operators $\mathbf{F}$, $\mathbf{G}$, and the binary connectives $\wedge, \vee$, defined by $\mathbf{F}(f) \equiv (\text{True}\ \mathbf{U}\ f)$, $\mathbf{G}(f) \equiv \neg\mathbf{F}(\neg f)$, $(f \vee g) \equiv (\neg f \supset g)$, $(f \wedge g) \equiv \neg(f \supset \neg g)$.

A finite-state Buchi automaton[1] A on infinite strings is 5-tuple $(Q, \Delta, \delta, I, F)$, where Q is a finite set of automaton states, Δ is a finite set of input symbols, $\delta : Q \times \Delta \to 2^Q$ is the transition function that given a state and an input symbol specifies the set of possible next states, $I \subseteq Q$ is the set of *initial* states, and $F \subseteq Q$ is the set of *final* states. We assume that for each $q \in Q$ and $t \in \Delta$, $\delta(q, t)$ is nonempty.

For any finite sequence σ, let length(σ) denote the number of elements in σ, and for an infinite sequence σ, let length(σ) be ω. Suppose A is an automaton and $(t_0, t_1, \ldots)$ is a finite or infinite sequence of symbols from Δ. Then a *run* of A on t is a sequence $(r_0, r_1, \ldots)$ of automaton states satisfying the following property: if length(t) is finite, then length(r) = length(t) + 1; otherwise, length(r) is ω; and $\forall i\ 0 \leq i < \text{length}(t)$, $r_{i+1} \in \delta(r_i, t_i)$. A run is said to be *accepting* if it has a final state occurring infinitely often. The automaton A is said to *accept* t if there is an accepting run of A on t starting with an initial state. (Note that under this definition, an automaton can accept only infinite strings.)

For an automaton A, we let $|A|$ denote the number of its states. We assume that the definitions of automata are given using a fixed encoding, and we let size(A) denote the length of this encoding for A. The connection between automata and temporal logic has been investigated in [6], [28], and [34]. Throughout the paper, we make use of the following fact that has been established in these papers: Corresponding to every PTL formula f, there is an automaton A_f with input alphabet $2^{|f|}$ that accepts exactly the set of sequences that satisfy f, and in addition $|A_f| = O(2^{|f|})$ and size(A_f) $= O(4^{|f|})$.

2.2. Model Checking Using Automata. The term, *model checking*, has been used in the literature to refer to an algorithmic approach for showing that a concurrent program satisfies a specification given in a formal system such as temporal logic or automata. Many different model-checking algorithms have been presented in the literature. Specifically, the papers [16, 29, 33] present such algorithms when the specification is given in PTL. Of these algorithms,

[1] From here onwards, the term, *automaton*, refers to a Buchi automaton.

the automata theoretic approach advocated in [33] is of particular interest to us. In this approach, the following method is used to check that all the executions of a finite-state concurrent program satisfy a PTL specification f. First, the automaton $A_{\neg f}$ corresponding to the specification $\neg f$ is obtained. Next, the program is modeled as an automaton P. If the program is given as a state graph, then, roughly speaking, this state graph can be considered as the automaton. Next, it is checked that there is no string that is accepted both by P and $A_{\neg f}$. To accomplish this, another automaton B is constructed; intuitively, B is a cross product of P and $A_{\neg f}$. B accepts exactly those strings that are accepted by both P and $A_{\neg f}$. The construction of B is such that $|B| = O(|P| \cdot |A_{\neg f}|)$, and verification is accomplished by showing that B does not accept any string. This whole procedure can be accomplished in time $O(\text{size}(P) \cdot \text{Size}(A_{\neg f}))$.

We consider systems with an arbitrary number of processes. Since all the possible behaviors of such systems cannot be modeled by a finite-state system, we cannot use the above approach for verification. However, we do use some of the ideas of this approach in certain parts of our algorithms.

2.3. Processes and Computations. We now introduce *process definitions*, which define the operational behavior of our systems. A *communication alphabet* Σ is a set of symbols that is the union of mutually disjoint sets $\Sigma^+, \Sigma^-, \{\epsilon\}$, where Σ^+ is a set of symbols called *actions*, and Σ^- consists of the complements of the actions, $\Sigma^- = \{\bar{c} : c \in \Sigma^+\}$. For notational convenience, for any $e = \bar{c} \in \Sigma^-$, we let $\bar{e} = c$. Each member of Σ is called a *communication symbol*. We say that two communication symbols c, c' are *complementary* if it is the case that $\bar{c} = c'$. A process definition U is a 4-tuple (S, R, I, Φ) where S is a finite set of states called process states, $R \subseteq S \times S \times \Sigma$ is a set of transitions, $I \subseteq S$ is the set of initial states, and $\Phi : S \to 2^{\mathcal{P}}$ is a function that associates with each state in S a set of atomic propositions that are true in that state. Intuitively, a process definition specifies the behavior of one or more processes in a system. Transitions of the form $(s, s', \epsilon) \in R$ are called *internal* transitions, and transitions of the form $(s, s', c) \in R$ where $c \neq \epsilon$ indicate possible communication between processes. For a process definition U, we let $|U|$ denote the number of states of the process, that is, $|U| = |S|$. For the decision procedures, we assume that process definitions are supplied using a fixed encoding. For a process definition U, we let size(U) denote the length of the encoding of U.

A *system of processes* M is an n-tuple $(U_1, \ldots, U_n)$ of process definitions. In our development, the intuitive notion of an individual process in a system is captured formally by its index in a system. Thus, we say that a *process* in M is a natural number i such that $1 \leq i \leq n$. Consider a system $(U_1, \ldots, U_n)$ of processes and let $U_i = (S_i, R_i, I_i, \Phi_i)$ for $1 \leq i \leq n$. A *global state* of the above system is an element of $S_1 \times S_2 \times \cdots \times S_n$, that is, a global state is an n-tuple of process states. For a global state δ and for $1 \leq i \leq n$, we let $\delta[i]$ denote the ith component of δ. Intuitively, $\delta[i]$ is the state of process i in the global state δ. The global state δ is an *initial* global state if $\delta[i] \in I_i$ for all i such that $1 \leq i \leq n$. We say that processes i and j can *communicate* (or *synchronize*) in the global state δ if there exist $c \in \Sigma$ where $c \neq \epsilon$, $x \in S_i$ and $y \in S_j$ such that $(\delta[i], x, c) \in R_i$ and $(\delta[j], y, \bar{c}) \in R_j$. We say that process i is *enabled* in the global state δ if either it can communicate with another process

in δ, or for some $x \in S_i$, $(\delta[i], x, \epsilon) \in R_i$. A global state is called a *deadlock state* if no process is enabled in it. Let δ and γ be two global states. We say that γ can be reached from δ by an internal transition of process i if $(\delta[i], \gamma[i], \epsilon) \in R_i$ and for $1 \leq j \leq n$, $j \neq i$, $\delta[j] = \gamma[j]$. We say that γ can be reached from δ by communication between processes i and j (for $i \neq j$) if for some $c \in \Sigma$, $(\delta[i], \gamma[i], c) \in R_i$, $(\delta[j], \gamma[j], \bar{c}) \in R_j$ and for all $k \neq i, j$, $1 \leq k \leq n$, $\delta[k] = \gamma[k]$. We say that γ can be reached from δ by a transition of process i if γ can be reached from δ by an internal transition of process i or by communication between process i and some other process. Note that for a pair of global states, there can be more than one way that the second state can be reached from the first state. We say that γ can be reached from δ in one computation step if γ can be reached from δ by an internal transition of a process or by communication between a pair of processes.

A *labeled computation sequence* C of a system of processes is a sequence of pairs $(\psi_0, \sigma_0), (\psi_1, \sigma_1), \ldots$ such that for $i \geq 0$, σ_i is a global state, ψ_{i+1} is a set of processes containing one or two processes, and such that for all i such that $0 < i \leq \text{length}(C) - 1$,

if $\psi_i = \{j\}$, then σ_i can be reached from σ_{i-1} by an internal transition of process j,

if $\psi_i = \{j, k\}$ $(j \neq k)$, then σ_i can be reached from σ_{i-1} by communication between j and k.

A finite or infinite sequence of global states $\sigma_0, \sigma_1, \ldots$ is called a *computation sequence* iff there exists a sequence $\psi_0, \psi_1, \ldots$ such that $(\psi_0, \sigma_0), (\psi_1, \sigma_1), \ldots$ is a labeled computation sequence. We define a *computation* to be a computation sequence that satisfies certain conditions on its beginning and ending. A computation begins in an initial global state and either ends in a deadlock state or is infinite. We call finite computations *deadlocked computations*.

We say that a labeled computation sequence $(\psi_0, \sigma_0), (\psi_1, \sigma_1), \ldots$ is *fair* if for each process i the following property holds: If i is enabled infinitely many times in the sequence, then there are infinitely many values of k such that $i \in \psi_k$. A computation sequence $\sigma_0, \sigma_1, \ldots$ is said to be fair if there exist $\psi_0, \psi_1, \ldots$ such that $(\psi_0, \sigma_0), (\psi_1, \sigma_1), \ldots$ is a fair labeled computation sequence. Note that all finite computation sequences are fair.

If $\sigma = \sigma_0, \sigma_1, \ldots$ is a sequence of global states, then we define the *execution of process i in the sequence* σ to be the projection of σ onto the ith coordinate, that is, the sequence $\sigma_0[i], \sigma_1[i], \ldots$. We define the *execution sequences* (resp., *executions*) of a single process in a system to be the projections of the computation sequences (resp., computations) of the system onto a single process. That is, a sequence $t = t_0, t_1, \ldots$ of states of the ith process definition is an execution sequence of process i if there exists a computation sequence of the system $\sigma_0, \sigma_1, \ldots$ where for $j \geq 0$, $t_j = \sigma_j[i]$. Moreover, if t is an execution of process i in the system then it begins in an initial state and is infinite or is a projection of a deadlocked computation. We say that t is a *fair* execution (sequence) of process i if it is a projection of a fair computation (sequence). For a PTL formula f, we say that the execution (sequence) t satisfies f iff the ω-sequence $\Phi_i(t_0), \Phi_i(t_1), \ldots$ satisfies f.

Throughout the paper, we use certain notational conventions for sequences. A single element, taken as a sequence, denotes a sequence of length 1. If $\alpha = \alpha_0, \ldots, \alpha_n$ is a finite sequence and $\beta = \beta_0, \beta_1, \ldots$ is a finite or infinite sequence, then $\alpha\beta$ denotes the *concatenation* of α and β, which is $\alpha_0, \ldots, \alpha_n, \beta_0, \beta_1, \ldots$; if the last element in α is the same as the first element in β, then $\alpha \bullet \beta$ denotes the *dot concatenation* of α and β, which is the result of combining the sequences without repeating the middle element, that is $\alpha_0, \ldots, \alpha_n, \beta_1, \beta_2, \ldots$. If α is a finite sequence, then α^n denotes the result of concatenating α n times, and α^ω denotes the result of concatenating α infinitely. Similarly, if α is a finite sequence whose first and last elements are the same, then $\alpha^{\bullet n}$ and $\alpha^{\bullet \omega}$ denote the results of dot concatenating α n times and infinitely.

For most of the paper, we consider model checking in which a property is defined to hold for a system provided it holds for only the infinite computations. We show how to handle deadlocked computations in Section 3.4. Henceforth, unless otherwise stated the term, *computation*, always refers to an infinite computation.

3. *Model Checking for Systems with a Control Process and Many User Processes*

In this section, we consider the systems of processes that contain a unique control process and many user processes with identical definitions. We believe that many resource allocation algorithms, network protocols, and mutual exclusion algorithms can be handled in this model at a certain level of abstraction. In Section 5, we show that the algorithm for mutual exclusion on rings considered in [4] can be handled in this model.

As in Section 2, let Σ be the communication alphabet containing actions, complements of actions, and the special symbol ϵ. Let $C = (S_C, R_C, I_C, \Phi_C)$, and $U = (S_U, R_U, I_U, \Phi_U)$ be the control and user process definitions.

For any $n \geq 0$, let $C \times U^n$ denote the system of processes $(C, U, \ldots, U)$ where the process definition U is repeated n times. In the system $C \times U^n$, we call process 1 the control process and processes $2, 3, \ldots, n+1$ user processes. Let Control-Exec$(C, U) = \{t$: for some $n \geq 0$, t is an execution of the control process in an infinite computation of $C \times U^n\}$. Similarly, let User-Exec(C, U) denote the set $\{t$: for some $n > 0$, t is an execution of a user process in an infinite computation of $C \times U^n\}$. The *model-checking problem* for the control process consists of checking if every execution in Control-Exec(C, U) satisfies a given PTL formula. Similarly, the model-checking problem for the user process consists of checking if every execution in User-Exec(C, U) satisfies a given PTL formula.

Let Fair-Control-Exec$(C, U) = \{t$: for some $n \geq 0$, t is an execution of the control process in a fair infinite computation of the system $C \times U^n\}$. Similarly, let Fair-User-Exec$(C, U) = \{t$: for some $n \geq 0$, t is an execution of a user process in a fair infinite computation of the system $C \times U^n\}$. The model-checking problem for fair computations of the control process consists of checking if every execution in Fair-Control-Exec(C, U) satisfies the given PTL formula. Similarly, we define the model-checking problem for fair computations of a user process.

In Section 3.2, we present algorithms for the model-checking problem for the control process and the user process. In Section 3.3, we give algorithms for

these problems for fair computations. In Section 3.4, we consider model checking for an extended set of computations that includes deadlocked computations.

Let f be the PTL specification. In order to check if all executions in Control-Exec(C, U) satisfy f, we verify that no execution in Control-Exec(C, U) satisfies $\neg f$. For this purpose, we construct the automaton $A_{\neg f}$, which accepts exactly the set of strings that satisfy $\neg f$. The input symbols for $A_{\neg f}$ are subsets of $\mathscr{P}$ (the set of atomic propositions). Each input symbol of $A_{\neg f}$ specifies which atomic propositions are true at the next instance.

The next step in developing the decision procedures is to model the executions of the control process and the runs of $A_{\neg f}$ on these executions by a *Vector Addition System with States* (VASS). A VASS is a slightly different formalism from Vector Addition Systems [11] and Petri nets [23]; however, all the three formalisms are equally powerful. A VASS of dimension m is a finite labeled directed graph in which the label of each edge is a vector of m integers. Let $\mathbf{Z}$ and $\mathbf{N}$ be the set of integers and the set of nonnegative integers respectively. Formally, a VASS G of dimension m is a pair (V, E) where V is a finite set of states and $E \subseteq V \times V \times \mathbf{Z}^m$ is a finite set of transitions. We can consider G to be a labeled directed graph in which there is an edge from s to s' with label $\vec{a}$ iff $(s, s', \vec{a}) \in E$. A *configuration* of G is a pair $c = (s, \vec{a})$ where $s \in V$ and $\vec{a} \in \mathbf{N}^m$. We call $\vec{a}$ the *configuration vector* of c. Note that all the components of a configuration vector are nonnegative. For configurations $d = (s, \vec{a})$, $d' = (s', \vec{c})$ of G, let $Tr(d, d')$ denote the triple $(s, s', \vec{c} - \vec{a})$. If $Tr(d, d')$ is a transition of G, then we say that the configuration d' can be *reached from d by the transition* $Tr(d, d')$. A *path* of G is a sequence of configurations $c_0, c_1, \ldots,$ such that for all $i \geq 0$, $Tr(c_i, c_{i+1})$ is a transition of G. A configuration d is said to be *reachable* from c in G if there is a path of G starting from c and ending in d. A path of finite length is said to be a *cycle* if its first and last configurations are the same and it is of length at least two. For any m-vector $\vec{a}$, we let $\vec{a}[i]$ denote the ith component of the vector. For any configuration $c = (s, \vec{a})$, we let state(c) denote the state s and vec(c) denote the vector $\vec{a}$. Similarly, for any transition $t = (s, s', \vec{a})$, we let source(t), target(t), and vec(t) respectively, denote the states s, s', and the vector $\vec{a}$. For any configuration $c = (s, \vec{a})$, let weight(c) denote the sum of all coordinates of $\vec{a}$.

We define a *family* to be a set of systems. We write $\{C \times U^n\}$ to denote the family consisting of all systems of the form $C \times U^n$, for $n \geq 0$. In order to avoid confusion, at some places, we refer to the members of R_C, R_U as process transitions.

Consider a single system $C \times U^n$ for $n \geq 0$ in a family. As far as the control process is concerned, at any step of a computation of the system, the information regarding the user processes can be represented by a vector of integers $\vec{a}$, where $\vec{a}[i]$ is the number of user processes in the ith user state. The reason we can use this representation is that the behavior of a user process depends only on its state and not on its index. From this, it follows that the behavior of the control process can be modeled by a VASS of dimension m, where m is the number of states in U. In fact, we construct a VASS that simultaneously models the executions of the control process and the runs of the automaton $A_{\neg f}$ on these executions.

3.1. CONSTRUCTION OF THE VASS. Appendix A gives a detailed construction of the VASS VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) = (V, E) for generic process definitions $\mathscr{C} = (S_{\mathscr{C}}, R_{\mathscr{C}}, I_{\mathscr{C}}, \Phi_{\mathscr{C}})$, $\mathscr{U} = (S_{\mathscr{U}}, R_{\mathscr{U}}, I_{\mathscr{U}}, \Phi_{\mathscr{U}})$, and the automaton $\mathscr{A} = (Q, \Delta, \delta, J, F)$ where $\Delta = 2^{\mathscr{P}}$ is the input alphabet of the automaton. Let $S_{\mathscr{U}} = \{u_1, u_2, \ldots, u_m\}$. In this section, we prove some properties of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) which will be used later. Consider the family of systems $\mathscr{G} = \{\mathscr{C} \times \mathscr{U}^n : n \geq 0\}$. Let gstates($\mathscr{G}$) be the set of all global states of systems in $\mathscr{G}$. In the remainder of Section 3.1, the term, *global state*, refers to any element in gstates($\mathscr{G}$). The set V is partitioned in to three disjoint sets $(S_{\mathscr{C}} \times Q)$, V_I, $\{s_0\}$ where the states in $S_{\mathscr{C}} \times Q$, V_I are called *proper states* and *intermediate states*, respectively, and the state s_0 is called the *initial state*. A configuration $(s, \vec{v})$ is called a *proper configuration*, (resp., an *intermediate configuration*) if s is a proper state (resp., an intermediate state). The configuration $(s_0, 0)$ is called the initial configuration. A path containing a proper configuration is called a *proper path*.

We define a relation *represents* between the proper configurations of *VS*($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) and gstates($\mathscr{G}$) as follows. Recall that for a global state σ, $\sigma[i]$ is the ith component of σ, that is, the state of process i. For a proper configuration $c = (s, \vec{v})$ where $s = (b, q)$, and a global state σ of a system $\mathscr{C} \times \mathscr{U}^n$, c *represents* σ iff $b = \sigma[1]$ and for all $i = 1, 2, \ldots, m$, $\vec{v}[i]$ is the number of distinct values of j such that $2 \leq j \leq n + 1$ and $\sigma[j] = u_i$. That is, b is the state of the control process in σ and for $i = 1, \ldots, m$, $\vec{v}[i]$ is the number of processes in the user state u_i in the global state σ. Note that for each proper configuration c, there can be more than one global state σ such that c represents σ; similarly, for each global state σ, there can be more than one configuration c that represents σ, corresponding to different values of q.

We say that a sequence of proper configurations $c_0, c_1, \ldots$ represents a sequence of global states $\sigma_0, \sigma_1, \ldots$ if c_i represents σ_i for each $i \geq 0$. For any path π of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$), let *reduced-path*(π) be the sequence of proper configurations appearing in π. Recall that a global state σ is an initial global state if the state of each process in σ is an initial state. The operation of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) can be described intuitively, as follows: The VASS will start in the initial state from which it nondeterministically chooses a configuration that represents an initial global state of a system $\mathscr{C} \times \mathscr{U}^n$ for some $n \geq 0$. The VASS chooses an initial state of a system by looping around in its initial state. From this point onwards, further transitions of the VASS take it along a path π such that reduced-path(π) represents a computation of the system S.

For any proper path π, let *first-proper*(π) denote the first proper configuration to appear in π. For any two proper configurations $d = (s, \vec{v})$, $d' = (s', \vec{w})$ where $s = (a, q)$, $s' = (a', q')$, we say that d' is an $\mathscr{A}$-successor of d if $q' \in \delta(q, \Phi_{\mathscr{C}}(a'))$. The following lemma gives some properties of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$), whose construction is given in Appendix A.

LEMMA 3.1. *Let π be a proper path of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$).*

(*a*) *Every configuration appearing after first-proper(π) is either a proper configuration or an intermediate configuration. Proper configurations and intermediate configurations alternate in π after first-proper(π).*

(*b*1) *If π starts with the initial configuration, then all the global states represented by first-proper(π) are initial global states.*

(*b*2) *For every initial global state* σ, *there exists a finite path* π' *starting with the initial configuration and ending with a proper configuration that represents* σ, *which is also the first proper configuration in* π'.

(*c*) *If* π *starts with the initial configuration and reduced-path*$(\pi) = c_0, c_1, \ldots,$ *where for all* $i \geq 0$, $c_i = (s_i, \vec{v}_i)$ *and* $s_i = (a_i, q_i)$, *then for some initial state* r_0 *of* $\mathscr{A}$, $r_0, q_0, q_1, \ldots, q_i, \ldots$ *is a run of* $\mathscr{A}$ *on* $\Phi_{\mathscr{C}}(a_0), \Phi_{\mathscr{C}}(a_1), \ldots$.

PROOF

Part (a) follows from the following facts: For any transition $(s, s', \vec{v})$ of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$), if s is a proper state, then s' is an intermediate state; if s is an intermediate state, then s' is a proper state.

Parts (b1), (b2), and (c) follow from the following facts: For any transition from state s_0, that is, for any transition of the form $(s_0, s, \vec{v})$, either $s = s_0$ or s is a proper state. If $s = s_0$, then such a transition increments the ith coordinate of the component vector corresponding to some initial state $u_i \in I_{\mathscr{U}}$. For every initial state $u_i \in I_{\mathscr{U}}$, there is a transition of this kind in E. If in the transition $(s_0, s, \vec{v})$, state s is a proper state and $s = (a, q)$, then for some initial state r_0 of $\mathscr{A}$, $q \in \delta(r_0, \Phi_{\mathscr{C}}(a))$, and $\vec{v}$ is the zero vector. If d, c, d' is a path of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) where d, d' are proper configurations and c is an intermediate configuration, then d' is an $\mathscr{A}$-successor of d. □

In Appendix A, we have classified certain transitions of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) as transitions from a pair of states and certain transitions as internal transitions from a single state. The following lemma, which is immediate from the definition of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$), gives some technical properties that will be used later.

LEMMA 3.2. *Let* c, c' *be proper configurations of* $G = VS(\mathscr{C}, \mathscr{U}, \mathscr{A})$ *such that* c' *is an* $\mathscr{A}$*-successor of* c, *and let* σ *be any global state that* c *represents.*

Then some global state σ' *represented by* c' *can be reached from* σ *in one computational step iff there is an intermediate configuration* d *such that* c, d, c' *is a path of* G. *In addition,* σ' *can be reached from* σ *by communication between processes* i, j (*respectively, by an internal transition of process* i) *iff for some intermediate configuration* d, c, d, c' *is a path of* G *and* $Tr(c, d)$ *is a communication transition of* G *from the pair of states* $(\sigma[i], \sigma[j])$ (*respectively,* $Tr(c, d)$ *is an internal transition of* G *from the state* $\sigma[i]$).

3.2. MODEL CHECKING WITHOUT FAIRNESS. Now we present the model-checking algorithms for the control and user processes. First, we give the algorithm for the control process and later show that the model-checking problem for the user process can be reduced to that of the control process.

Now we describe the various steps of our algorithm. We are given process definitions C and U, and a PTL formula f. Let $G = (V, E)$ be the VASS VS($C, U, A_{\neg f}$). We say that a proper configuration $c = (s, \vec{v})$ of G, where $s = (a, q)$, is a *final configuration* if q is a final state of the automaton $A_{\neg f}$.

The following lemma relates the executions in Control-Exec(C, U) that satisfy $\neg f$ to certain infinite paths of G.

LEMMA 3.3. *The following are equivalent*:

(*a*) *There is an execution in Control-Exec*(C, U) *that satisfies* $\neg f$.
(*b*) *G has a an infinite path starting from the initial configuration and containing infinitely many final configurations.*

PROOF

($a \Rightarrow b$). Assume (a). Let $e = e_0, e_1, \ldots$ be an execution in Control-Exec(C, U) such that $\Phi_C(e)$ satisfies $\neg f$.[2] Let $q = q_0, q_1, \ldots$ be an accepting run of $A_{\neg f}$ on $\Phi_C(e)$. Also, let $\sigma = \sigma_0, \sigma_1, \ldots$ be a computation of the system $C \times U^n$, for some $n \geq 0$, such that the execution of the control process in σ is e. For all $i \geq 0$, let $c_i = (s_i, \vec{v}_i)$ be the proper configuration of G such that $s_i = (e_i, q_{i+1})$ and c_i represents σ_i. From Lemma 3.2, we see that, for all $i \geq 0$, there exists an intermediate configuration d_i such that c_i, d_i, c_{i+1} is a path of G. From this, we see that there exists an infinite path π of G starting with c_0 such that reduced-path$(\pi) = c_0, c_1, \ldots$. Since for infinitely many values of i, q_i is a final state of $A_{\neg f}$, it follows that π contains infinitely many final configurations. Using (b2) of Lemma 3.1, we see that there is a path $\alpha\pi$ starting from an initial configuration and containing infinitely many final configurations.

($b \Rightarrow a$). Assume (b). Let π be an infinite path of G that starts with the initial configuration and that contains infinitely many final configurations. Let $c_0, c_1, \ldots$ be the sequence of proper configurations appearing in π; clearly, all the final configurations appearing in π also appear in the above sequence. For all $i \geq 0$, let $c_i = (s_i, \vec{v}_i)$ where $s_i = (e_i, q_i)$. Using (b1) of Lemma 3.1, we see that there exists an initial global state σ_0 such that c_0 represents σ_0. Using (a) of Lemma 3.1, we see that, for each $i \geq 0$, c_i and c_{i+1} are separated in π by an occurrence of an intermediate configuration d_i. Now using Lemma 3.2 inductively, it can be shown that there exists a computation $\sigma = \sigma_0, \sigma_1, \ldots$ where for all $i \geq 0$, c_i represents σ_i. Now using (c) of Lemma 3.1, we see that for some initial state r_0 of $A_{\neg f}$, $r_0, q_0, q_1, \ldots, q_i, \ldots$ is a run of $A_{\neg f}$ on $\Phi_C(e)$ where e is the execution of the control process in σ. Clearly, this is an accepting run as it contains a final state of $A_{\neg f}$ infinitely often. Now (a) follows from this. □

LEMMA 3.4. *G has an infinite path starting from the initial configuration and containing infinitely many final configurations iff G has a finite path of the form* $\alpha\beta$ *with the following properties*:

(*a*) α *starts in the initial configuration*;
(*b*) β *is a cycle*;
(*c*) *a final configuration appears in* β.

PROOF. Assume that G has an infinite path starting from the initial configuration and containing infinitely many final configurations. Let $c = c_0, c_1, \ldots$ be such a path. By the construction of G, this path must contain a proper configuration. For $j \geq 0$, let $c_j = (s_j, \vec{a}_j)$. Recall that weight(c_j) = sum of the components of $\vec{a}_j$. Let c_p be the earliest proper configuration in the path. From the construction of G, for any pair of configurations c, c' where c is a proper configuration and such that c' can be reached from c, the following property

[2] Recall that $\Phi_C(e) = \Phi_C(e_0), \Phi_C(e_1), \ldots$.

holds: weight(c') $\leq$ weight(c) and in addition, if c' is a proper configuration, then weight(c) = weight(c'). Hence, for all $k \geq p$, weight(c_k) $\leq$ weight(c_p). Clearly, the total number of distinct configurations appearing in c is finite. Thus, there exists an instance j with the property that the only configurations that appear beyond c_j are those that appear infinitely often in c. From this observation it is easily seen that there are finite paths α, β such that properties (a), (b), and (c) are satisfied.

Now assume that there is a finite path $\alpha\beta$ that satisfies the conditions (a), (b), and (c). By repeating the path β infinitely many times, we get the infinite path $\alpha\beta^{\cdot\omega}$ with the required property. □

Let $|V| = p$; that is, p is the number of states of G. Recall that for any process definition W, $|W|$ is the number of states of W and size(W) is the length of the encoding of W. Let $|U| = m$. The following lemma is proved on the same lines as the corresponding proofs given in [24] and [26]. Details of the proof are given in Appendix B.

LEMMA 3.5. *There exists a finite path of G of the form $\alpha\beta$ where α, β satisfy the conditions (a), (b), and (c) of Lemma 3.4 iff there exists such a path of length $O(2^{k_1 \cdot p \cdot \log(p) 2^{k_1 m \log(m)}})$ where k_1 is a constant.*

Note that the number of transitions of C is independent of the number of states of C, and as a consequence size(C) can be arbitrarily larger than $|C|$.

THEOREM 3.6. *The model-checking problem for the control process with input a temporal logic specification f, and process definitions C and U can be solved in time $h(size(C), size(U), 2^{|f|}) + O(2^{g(|f|, |C|, |U|)})$, where $h(x_1, x_2, x_3)$ is a polynomial function in x_1, x_2, x_3, and $g(|f|, |C|, |U|) = O(|C|^{k_1} \cdot 2^{k_2 \cdot (|f| + |U| \cdot \log(|U|))})$, where k_1 and k_2 are constants.*

PROOF SKETCH. In the first step, the algorithm obtains the automaton $A_{\neg f}$ and then constructs the VASS $G = (V, E) = \text{VS}(C, U, A_{\neg f})$. The automaton $A_{\neg f}$ can be constructed in time bounded by $2^{k \cdot |f|}$ for some constant k. From this, it is not difficult to see that the VASS G can be constructed in time $h(\text{size}(C), \text{size}(U), 2^{|f|})$ for some polynomial h. Let $p = |V|$ and $m = |U|$. In the second step, the algorithm checks that there is no finite path π of G which is of length $O(2^{k \cdot p \cdot \log(p) \cdot 2^{k m \log(m)}})$ where k is the constant of Lemma 3.5, such that π is of the form $\alpha\beta$ where α, β satisfy conditions (a), (b), (c) of Lemma 3.4. We can give a nondeterministic procedure for the second step that uses space $O(p \cdot \log(p) 2^{k \cdot m \cdot \log(m)})$. Using Savitch's result [27], we can obtain a deterministic procedure for the second step of the algorithm that uses space $O((p \cdot \log(p))^2 \cdot 2^{k \cdot m \cdot \log(m)})$ where k is a constant. In Appendix A, it has been shown that $p \leq (|C| \cdot |A_{\neg f}| m^2 + m + 2) + 1)$ and hence $p = O(|C| \cdot 2^{|f|} \cdot m^2)$. Substituting this value for p and simplifying it, it is easily seen that we can obtain a deterministic algorithm for the second step that uses space $g(|f|, |C|, |U|)$ and time $O(2^{g(|f|, |C|, |U|)})$. From this, it should be easy to see that we can get a model-checking algorithm that runs in the stated time and uses space $h(\text{size}(C), \text{size})U), 2^{|f|}) + g(|f|, |C|, |U|)$. □

The following corollary is a direct consequence of Theorem 3.6. It suggests a different approach to the model-checking problem, which consists of verifying

that the executions of the control process satisfy the given specification in all systems containing up to a certain bounded number of user processes.

COROLLARY. *The following are equivalent*:

(*a*) *All the executions in Control-Exec*(C, U) *satisfy* f.

(*b*) *For all* $n \leq 2^{g(|f|, |C|, |U|)}$, *all executions of the control process in the system* $C \times U^n$ *satisfy* f *where* $g(|f|, |C|, |U|) = O(|C|^{k_1} \cdot 2^{k_2 \cdot (|f| + |U| \cdot \log(|U|))})$ *and* k_1, k_2 *are constants.*

Proof Sketch. From Lemma 3.5, we see that if there exists an execution in Control-Exec(C, U) that does not satisfy f then there exists a path of length $2^{g(|f|, |C|, |U|)}$ in G that is of the form $\alpha\beta$ where α, β satisfy conditions (a), (b), (c) of Lemma 3.5. Clearly, for every configuration c in this path, all the components of vec(c) are bounded by $2^{g(|f|, |C|, |U|)}$. From this it follows that there exists an execution in Control-Exec(C, U) that satisfies $\neg f$ iff there exists $n \leq 2^{g(|f|, |C|, |U|)}$ such that some execution of the control process in the system $C \times U^n$ satisfies $\neg f$. The lemma follows from this observation. □

So far, we have considered the problem of checking whether all the executions of the *control* process satisfy a given specification. Now we investigate the problem of checking whether all the executions of a user process satisfy a given specification. We show that this problem can be reduced to the previous problem, and thus, the model-checking algorithm for the control process can also be used for the user process. We again assume that $C = (S_C, R_C, I_C, \Phi_C)$, $U = (S_U, R_U, I_U, \Phi_C)$ are respectively the definitions of the control and user processes, and f is the given PTL specification.

Now we present the reduction procedure. First, we construct a process definition C', which roughly speaking simulates C and U. The states of C' are pairs of the form (s, t) where s is a state of C and t is a state of U. Now we consider the system consisting of the control process whose definition is given by C' and an arbitrary number of user processes whose definition is given by U. The executions of a user process in the original system are obtained by considering the executions of the control process in the new system and projecting them onto the user state component. The process definition $C' = (S', R', I', \Phi')$ is defined as follows: The set of states $S' = \{(s, t) : s \in S_C \text{ and } t \in S_U\}$. The set of transitions R' contains exactly the following elements. For every transition $(s, s', c) \in R_C$ and every state $t \in S_U$, the transition $((s, t), (s', t), c)$ is in R', and similarly, for every transition $(t, t', e) \in R_U$ and state $s \in S_C$, the transition $((s, t), (s, t'), e)$ is in R'. In addition, for every pair of transitions $(s, s', c) \in R_C$, $(t, t', c') \in R_U$, where c and c' are complementary communication symbols, the transition $((s, t), (s', t'), \epsilon)$ is also in R'. This transition corresponds to synchronization between the original control process and the single-user process that we are considering. The set of initial states $I' = \{(s, t) : s, t,$ respectively, are initial states of C and $U\}$. For each state $(s, t) \in S'$, $\Phi_{C'}((s, t)) = \Phi_U(t)$, that is, the set of atomic propositions defined to be true in the state (s, t) is exactly the set of atomic propositions that are true in the user state t.

Define user(e), where e is a sequence of states of C', to be the sequence of states of U that is the projection of e onto the user-state component. Let $\text{Ex}(C, U) = \{\text{user}(e) : e \in \text{Control-Exec}(C', U)\}$. The following lemma states

that $\mathrm{Ex}(C, U)$ is exactly the set of executions of a user process in a system consisting of a control process whose definition is given by C and arbitrary number of user processes whose definition is given by U.

LEMMA 3.7. *User-Exec*(C, U) = *Ex*(C, U).

PROOF. First we prove that User-Exec$(C, U) \subseteq \mathrm{EX}(C, U)$. Let $u \in$ User-Exec(C, U). For some $n > 0$, u is the execution of a user process in the system $C \times U^n$. Without loss of generality, assume that u is the execution of the first user process in the computation $\sigma_0, \sigma_1, \ldots$ of the system $C \times U^n$. For all $i \geq 0$, let σ_i' be the global state of the system $C' \times U^{n-1}$ obtained by combining the states of the control process and the first user process in σ_i to form a state of C'. It should be clear that $\sigma_0', \sigma_1', \ldots$ is a computation of the system $C' \times U^{n-1}$. If e is the execution of C' in this computation, then user$(e) = u$. Hence, we see that User-Exec$(C, U) \subseteq \mathrm{Ex}(C, U)$.

Using similar arguments, it is easy to show that $\mathrm{Ex}(C, U) \subseteq$ User-Exec(C, U). □

The following theorem easily follows from Lemma 3.7 and from the way Φ' is defined.

THEOREM 3.8. *For C, U, and C' as specified above and for a given PTL formula f the following are equivalent*:

(*a*) *Every execution in User-Exec*(C, U) *satisfies* f.
(*b*) *Every execution in Control-Exec*(C', U) *satisfies* f.

To check for condition (a) of the above theorem, we construct C' and check for condition (b) of the above theorem. This can be done using the previously described model-checking algorithm for the control process.

3.3. MODEL CHECKING UNDER FAIRNESS. In concurrent systems, many times we are interested in verifying that a correctness property holds on all fair computations. This is especially the case for liveness properties. In this section, we consider the model-checking problem for fair computations of a system with a control process and many user processes. Let $C = (S_C, R_C, I_C, \Phi_C)$ and $U = (S_U, R_U, I_U, \Phi_C)$ be the control and user process definitions respectively where S_C and S_U are disjoint. Let $A_{\neg f} = (Q, \Delta, \delta, I, F)$ where Q is the set of the automaton states. Let $u_1, \ldots, u_m$ be the user states. We consider the problem of verifying that every execution in Fair-Control-Exec(C, U) satisfies a given PTL specification f. As in the previous section, from C, U, and $A_{\neg f}$ we construct the VASS $G = (V, E) = \mathrm{VS}(C, U, A_{\neg f})$. Recall that V is a disjoint union of the sets $(S_C \times Q), V_I, \{s_0\}$ where the members of $S_C \times Q$ are called proper states, those of V_I are called intermediate states and s_0 is called the initial state of G. Also, a configuration $(s, \vec{v})$ of G is called a proper configuration if s is a proper state. For a proper configuration $c = (s, \vec{v})$ where $s = (b, q)$, if σ is any global state such that c represents σ, then b is the state of the control process in σ and for each user state u_i there are $\vec{v}[i]$ user processes in state u_i in σ.

In order to develop the decision procedure for fair computations, we extend some of the definitions of Section 2 to paths of a VASS. We define a notion of

fairness for a VASS, and relate it to fair computation sequences of a system of processes. The notion of fairness in a VASS that we consider is different from other notions of liveness and fairness for Petri nets considered in literature (see [22] for definitions and references).

In Section 2, we defined what it means for a process i to be enabled in a global state σ. We now define a *process state* x to be enabled in a global state σ if there is an enabled process in state x in σ. For any proper configuration $c = (s, \vec{v})$ of G and for any state $x \in S_C \cup S_U$, we say that *there is a process in state x in c* if the following condition is satisfied: If $x \in S_C$, then $s = (x, q)$ for some $q \in Q$; if $x \in S_U$ and $x = u_i$, then $\vec{v}[i] > 0$. Now, note that for a proper configuration c and $x \in (S_C \cup S_U)$, there is a process in state x in c iff there is a process in state x in all the global states represented by c.

Let $t = (y, z, \vec{w})$ be a transition in E, and let $c = (s, \vec{v})$ be a configuration of G. We say that t *is enabled in* c if $y = s$ and $\vec{v} + \vec{w} \geq 0$. In Appendix A, we have classified certain transitions of E as communication transitions from (or to) a pair of states and certain transitions as internal transitions from (or to) a single state. We say that t is a *transition from (or to) state* x if t is a communication transition from (or to) a pair of states (x, y) or t is an internal transition from (or to) state x. For $x \in (S_C \cup S_U)$, we say that x *is enabled in* c if there exists a transition $t \in E$ from state x such that t is enabled in c. Note that if $t \in E$ is a transition from state x and t is enabled in c, then c has to be a proper configuration. As a consequence, for all $x \in S_C \cup S_U$ and all intermediate configurations c, state x is not enabled in c. From the construction of G, it follows that, for every $x \in S_C \cup S_U$, and every proper configuration c of G, state x is enabled in a proper configuration c iff state x is enabled in all the global states represented by c.

Recall that if c, d is a path of G, then $\mathrm{Tr}(c, d)$ denotes the transition by which d can be reached from c. Now, consider a path π of G and let c, d be any two consecutive configurations in π. If c is a proper configuration, then d is an intermediate configuration and for some $x \in S_C \cup S_U$, $\mathrm{Tr}(c, d)$ is a transition from x. We say that there is a *transition from state* x (respectively, *to state* x) in π if there exist consecutive configurations c_i, c_{i+1} in π such that $\mathrm{Tr}(c_i, c_{i+1})$ is a transition from state x (respectively, is a transition to state x).

Using these definitions, we can give a definition of fairness for paths of a VASS of the form $\mathrm{VS}(C, U, A_{\neg f})$. We say that an infinite path $c_0, c_1, \ldots$ of $G = \mathrm{VS}(C, U, A_{\neg f})$ is *fair* iff it is proper path and for each $x \in S_C \cup S_U$, if x is enabled infinitely often in the path, then for infinitely many values of i, $\mathrm{Tr}(c_i, c_{i+1})$ is a transition of G from state x. The following lemma relates the fair paths to fair computation sequences. Note that there can exist infinite paths α and β of G such that reduced-path(α) = reduced-path(β), and α is fair but β is not.

LEMMA 3.9. *Let C and U be process definitions, and α be an infinite sequence of proper configurations of $G = VS(C, U, A_{\neg f})$. Then,*

(i) there exists a fair path α' of G such that reduced-path(α') = α iff
(ii) there is a fair computation sequence σ of a system $C \times U^n$, for some n, such that α represents σ.

PROOF. Let $\alpha = \alpha_0, \alpha_1, \ldots$ be an infinite sequence of proper configurations of G.

(i) ⇒ (ii). Assume that α' is a fair path of G such that reduced-path(α') = α. Let n = weight(α_0). Now, we give a fair-labeled computation sequence (ψ_0, σ_0), $(\psi_1, \sigma_1), \ldots$ of the system $C \times U^n$. We denote the processes in the system $C \times U^n$ with integers $1, 2, \ldots, n, n+1$ with process 1 denoting the control process. In order to define the above-labeled computation sequence, we also define a sequence of queues of processes $q_0, q_1, \ldots$ where each q_i is a permutation of the sequence $1, 2, \ldots, n+1$. Intuitively, each q_i denotes the relative priorities of the processes. We define the above sequences inductively.

Let σ_0 be any global state represented by α_0, q_0 be any permutation of $1, 2, \ldots, n+1$, and ψ_0 be any set containing one or two process indices. Now, for some $j \geq 0$, assume that for all i such that $0 \leq i \leq j$, ψ_i, σ_i, q_i are defined. Now we show how to obtain ψ_{j+1}, σ_{j+1}, and q_{j+1}. For any $x \in S_C \cup S_U$, let $I_{x,j}$ be the set of processes in state x in the global state σ_j. From (a) of Lemma 3.1, we see that there is an intermediate configuration d_j such that $\alpha_j, d_j, \alpha_{j+1}$ appear consecutively in α'. Now let t, t' denote the transitions $\mathrm{Tr}(\alpha_j, d_j), \mathrm{Tr}(d_j, \alpha_{j+1})$, respectively. From the construction of G, it is seen that either case (a) or case (b), given below, holds: (a) t is a communication transition from some pair of states (x, y), and t' is a communication transition to some pair of states (x', y'); (b) t is an internal transition from some state x, and t' is an internal transition to some state x'. Assume that case (a) holds. Now let $\psi_{j+1} = \{r, r'\}$ where r is the earliest process on q_j among all processes in $I_{x,j}$, and r' is the earliest process on q_j among the process in $I_{y,j} - \{r\}$. Now using Lemma 3.2, we get a global state σ_{j+1} represented by α_{j+1} such that σ_{j+1} can be reached from σ_j by communication between the processes r, r'. Assume case (b). Let $\psi_{j+1} = \{r\}$ where r is the earliest process on q_j among all those in $I_{x,j}$. Using Lemma 3.2, we get a global state σ_{j+1} represented by α_{j+1} and such that σ_{j+1} can be reached from σ_j by an internal transition of process r. In both cases (a) and (b), let q_{j+1} be the queue obtained from q_j by moving the processes in ψ_{j+1} from their current positions in q_j to the end of the queue. Since α' is a fair path it should be easy to see that $(\psi_0, \sigma_0), (\psi_1, \sigma_1), \ldots$ is a fair-labeled computation sequence, and hence $\sigma_0, \sigma_1, \ldots$ is a fair computation; and α represents this computation.

(ii) ⇒ (i). Assume that $\sigma = \sigma_0, \sigma_1, \ldots$ is a fair-labeled computation sequence, and α represents σ. Clearly, there exists $\psi_0, \psi_1, \ldots$ such that $\sigma' = (\psi_0, \sigma_0), (\psi_1, \sigma_1), \ldots$ is a fair-labeled computation sequence. Now, using Lemma 3.2, it is easy to see that, for each $i \geq 0$, there exists an intermediate configuration d_i such that $\alpha_i, d_i, \alpha_{i+1}$ is a path of G such that for every $r \in \psi_{i+1}$, $\mathrm{Tr}(\alpha_i, d_i)$ is a transition of G from the state $\sigma_i[r]$. Clearly, $\alpha' = \alpha_0, d_0, \alpha_1, d_1, \ldots, \alpha_i, d_i, \alpha_{i+1}, \ldots$ is an infinite path of G. Now, we need to show that α' is a fair path. Let x be any state in $S_C \cup S_U$ such that x is enabled infinitely often in α', that is, for infinitely many values of i, x is enabled in α_i. From this, we see that there exists a process r such that for infinitely many values of i, $\sigma_i[r] = x$ and r is enabled in σ_i. Since σ' is fair, it is the case that for infinitely many values of i, $r \in \psi_i$. Hence, for infinitely many values of i, $\mathrm{Tr}(\alpha_i, d_i)$ is a transition from state x. Thus, α' is a fair path of G. □

We now give a complete decision procedure for model checking under fairness. The following lemma gives a necessary and sufficient condition for the existence of a fair execution of the control process that satisfies $\neg f$.

LEMMA 3.10. *The following are equivalent:*

(*i*) *For some n, in the system* $C \times U^n$, *there is an infinite fair execution e of the control process such that* $\Phi_C(e)$ *satisfies* $\neg f$.

(*ii*) *The VASS* $G = VS(C, U, A_{\neg f})$ *has a fair path starting from the initial configuration and containing infinitely many final configurations.*

PROOF

(i) ⇒ (ii). Assume (i). Let σ be a fair computation such that the execution of the control process in σ is e and $\Phi_C(e)$ satisfies $\neg f$ where $e = e_0, e_1, \ldots$ Let $q_0, q_1, \ldots$ be an accepting run of $A_{\neg f}$ on $\Phi_C(e)$ and $\alpha = \alpha_0, \alpha_1, \ldots$ be the sequence of proper configurations such that α represents σ, and for all $i \geq 0$, $\alpha_i = (s_i, \vec{v}_i)$ where $s_i = (e_i, q_{i+1})$. Clearly, α contains infinitely many final configurations. By Lemma 3.9, there exists a fair path α' starting from α_0 such that reduced-path(α') = α. Now, using (b2) of Lemma 3.1, it is easy to see that there exists an infinite path $\beta\alpha$ starting from the initial configuration. Clearly, this path is fair and contains infinitely many final configurations.

(ii) ⇒ (i). Assume (ii). Let α' be an infinite fair path containing infinitely many final configurations. Let α = reduced-path(α'). By Lemma 3.9, there exists a fair computation sequence σ such that α represents σ. Let $\alpha = \alpha_0, \alpha_1, \ldots$ where for all $i \geq 0$, $\alpha_i = (s_i, \vec{v}_i)$ and $s_i = (e_i, q_i)$. Let e denote the execution $(e_0, e_1, \ldots)$ of the control process in the above computation. By (c) of Lemma 3.1, for some initial state r_0 of $A_{\neg f}$, $r_0, q_0, q_1, \ldots, q_i, \ldots$ is a run of $A_{\neg f}$ on $\Phi_C(e)$. Clearly, this is an accepting run as it contains a final state of $A_{\neg f}$ appearing infinitely often. Hence, $\Phi_C(e)$ satisfies $\neg f$. Using (b1) of Lemma 3.1, it is easy to see that σ starts with an initial global state. □

THEOREM 3.11. *The problem of model checking under fairness is decidable.*

PROOF. The basic idea is to use the decidability of the reachability problem for a VASS[3] [12, 19] to determine whether the VASS $G = \text{VS}(C, U, A_{\neg f})$ has a fair path starting from the initial configuration and containing infinitely many final configurations. Given a VASS and two configurations, the reachability problem is to decide whether there is a path from the first configuration to the second.

In order to use the reachability problem, we show that the question of whether the VASS G has a fair path containing infinitely many final configurations can be reduced to the problem of checking whether G has a finite path with certain properties. We do this in two steps. In the first step, we find a characterization of a finite path that G has iff it has the required infinite path. In the second step, we find another characterization of the finite path that is equivalent to the first one, but that can be expressed as a reachability problem.

Consider a VASS defined by $\text{VS}(C, U, A_{\neg f})$, where C and U are process definitions. We will say that a finite path π of $\text{VS}(C, U, A_{\neg f})$ is *j-fair*, where j is a state of C or U, if either there is a transition in π from state j, or state j is never enabled in π. Recall that a cycle is a path of length at least two and that

[3] We can apply these results because of the equivalence of Petri nets and Vector Addition Systems to VASSes.

begins and ends in the same configuration. The first step of the proof is given by the following lemma.

LEMMA 3.12. *The VASS* $G = VS(C, U, A_{\neg f})$ *has a fair path starting from the initial configuration and containing infinitely many final configurations iff G has a finite path of the form* $\alpha \bullet \beta$ *satisfying the following properties*:

(*a*) α *starts in the initial configuration and ends in the first configuration of* β.
(*b*) β *is a cycle starting with a proper configuration.*
(*c*) *A final configuration appears in* β.
(*d*) *For all states* $j \in S_C \cup S_U$, β *is j-fair.*

PROOF. Assume that G has a fair path starting from an initial configuration and containing infinitely many final configurations. Let $c = c_0, c_1, \ldots$ be such a path. Using the same argument as in the proof of Lemma 3.4, we can show that there exists a number i, with the following properties: The only configurations that appear beyond c_i are those that appear infinitely often in c; for all $j \in S_C \cup S_U$, if j is enabled only a finite number of times, then it is never enabled beyond c_i. From the above observation, it should be clear that there exist finite paths α, β such that properties (a)–(d) are satisfied.

Now assume that there is a finite path $\alpha \bullet \beta$ that satisfies the conditions (a) thru (d) of the lemma. By repeating the path β infinitely many times we get the infinite path $\alpha \bullet \beta^{\bullet\omega}$, which has the required property. □

Next, we reformulate the conditions of Lemma 3.12 in such a way that we can use the reachability problem. It is straightforward to express conditions (a), (b), and (c) using the reachability problem for VASS. The difficulty comes with condition (d), which involves fairness. Intuitively, we cannot "force" a VASS to make a transition when a state is enabled and thus the fairness conditions cannot be checked directly.

To overcome this difficulty, we find a different characterization of the path β which implies that β can be repeated infinitely to give an infinite fair path. The new characterization is based on a condition that is called *separate fairness*. Intuitively, this condition is a property of the separate executions of the processes in a system, and can be tested more easily by a VASS. We define separate fairness for paths of $VS(C, U, A_{\neg f})$.

Previously, we defined a notion of a process state being enabled in a global state or a process state being enabled in a proper configuration. In order to define separate fairness, we need the notion of one process state enabling another process state. Let $C = (S_C, R_C, I_C, \Phi_C)$, $U = (S_U, R_U, I_U, \Phi_U)$ be process definitions where S_C, S_U are disjoint. Let $s, s' \in S_C \cup S_U$. We say that s *enables* s' iff there exist transitions (s, t, c), $(s', t', \bar{c}) \in R_C \cup R_U$. Note that for any global state of the system $C \times U^n$ such that processes i, j, respectively, are in states s, s' in σ, another global state σ' can be reached from σ by communication between i, j iff s enables s'.

We can now state the definition of separate fairness. For a state $j \in S_C \cup S_U$, let us say that a finite path π of $VS(C, U, A_{\neg f})$ that begins and ends with proper configurations is *separately fair with respect to j*, in short *j-sep fair*, if at least one of the following three conditions holds:

(1) There is a transition from state j in π.

(2) At both the beginning and end of π, there is no process in state j.
(3) The following two conditions hold:
  (3a) There is no internal process transition defined from state j;
  (3b) For each state k that enables state j, at both the beginning and end of π there is no process in state k, and there is no transition from state k in π.

The following lemma relates the notions j-fair and j-sepfair.

LEMMA 3.13. *Let $G = VS(C, U, A_{\neg f})$. A cycle of the VASS G starting with a proper configuration is j-sepfair iff it is j-fair.*

PROOF. We use the following claim in the proof. The claim is obvious from the construction of G.

Claim. Let π be a cycle of G starting with proper configuration, and let $j \in S_C \cup S_C$. Then there is no process in state j in each proper configuration appearing in π iff at the beginning and end of π there is no process in state j and there is no transition from state j in π. □

We now prove the lemma. Assume that π is a cycle of G starting with a proper configuration. Suppose that π is j-fair. If Condition (1) holds, then there is a transition from state j in π, so π must be j-fair. If Condition (2) holds, then from the above claim, we see that either there is no process in state j in each of the proper configurations appearing in π, and hence state j is never enabled in π; or there is a transition from state j in π; clearly, both of the above cases imply that π is j-fair. Condition (3) and the above claim also imply that state j is never enabled in π, and so π must be j-fair.

Now suppose that π is a j-fair cycle of G. If there is a transition from state j in π, then π is j-sepfair by (1). Otherwise, if (1) does not hold, state j must never be enabled. This can happen if there is no process in state j in each of the proper configurations appearing in π (Condition (2)). Finally assume (1) does not hold and that there is a process in state j somewhere in π. Since π is a cycle, from the construction of G, it is easy to see that there must be a process in state j in every proper configuration appearing in π. Hence, state j is not enabled any where in π. As a consequence, it has to be the case that there is no internal process transition from state j and for each proper configuration c appearing in π, and for each state $k \in S_C \cup S_U$ such that k enables j, there is no a process in state k in the configuration c. Hence, using the claim, we see that Condition (3) must hold and π is j-sepfair. □

We can now reformulate the conditions of Lemma 3.12, using the notion of separate fairness.

COROLLARY 3.14. *The VASS $G = VS(C, U, A_{\neg f})$ has a path starting from the initial configuration and containing infinitely many final configurations iff G has a finite path of the form $\alpha \bullet \beta$ satisfying the following conditions:*

(a) α starts in the inital configuration and ends in the first configuration of β;
(b) β is a cycle starting with a proper configuration;
(c) a final configuration appears in β;
(d) for all states $j \in S_C \cup S_U$, β is j-sepfair.

We now turn attention to showing how the conditions of the corollary can be expressed as a reachability problem, in order to complete the proof of Theorem 3.11. The basic idea is that separate fairness can be tested by counting various kinds of transitions along the path.

Let $G = \mathrm{VS}(C, U, A_{\neg f})$. Let $c = (s, \vec{v})$ be a proper configuration of G. For a state $u_i \in S_U$, the number of processes in state u_i is $\vec{v}[i]$. For a state $b \in S_C$, the number of processes in state b is one if there is a process in state b in c, that is, $s = (b, q)$ for some state q of $A_{\neg f}$, and is zero, otherwise. For any $j \in S_C \cup S_U$ and for a finite path π of G, where $\pi = c_0, c_1, \ldots, c_n$, we define *the number of transitions leaving state j* to be the number of values of i such that $0 \leq i < n$ and $\mathrm{Tr}(c_i, c_{i+1})$ is a transition from state j.

Let us define the following functions from finite paths π of $G = (V, E) = (\mathrm{VS}(C, U, A_{\neg f})$ to natural numbers. To define these functions, we assume that the states in V are mapped to consecutive natural numbers starting from zero. For any $s \in V$, we let $\mathrm{nbr}(v)$ denote this number.

$$\text{init-state-nbr}(\pi) = \mathrm{nbr}(s) \text{ where } s = \text{state}(c)^4 \text{ and } c \text{ is the first configuration in } \pi.$$

$$\text{end-state-nbr}(\pi) = \mathrm{nbr}(s') \text{ where } s' = \text{state}(c') \text{ and } c' \text{ is the last configuration of } \pi.$$

$$\text{final-count}(\pi) = \text{number of final configurations in } \pi.$$

$$\text{transition-count}(\pi) = (\text{number of configurations in } \pi) - 1.$$

For each state $j \in S_C \cup S_U$, we define the following four functions for a path π beginning and ending with proper configurations:

$$\mathrm{init}_j(\pi) = \text{number of processes in state } j \text{ in the first configuration of } \pi.$$

$$\mathrm{end}_j(\pi) = \text{number of processes in state } j \text{ in the last configuration of } \pi.$$

$$\mathrm{exit}_j(\pi) = \text{number of transitions leaving state } j \text{ in } \pi.$$

We can reexpress the conditions of the corollary in terms of these functions, as (a′)–(d′) below. The condition (a′) is the same as condition (a).

Condition (b′), given below, needs a bit of explanation. By definition, a path β is a cycle if it begins and ends in the same configuration and is of length at least two. We will check whether the initial and final configurations of β are the same by checking that init-state-nbr(π) = end-state-nbr(π) and for all states $j \in S_U$, $\mathrm{init}_j(\beta) = \mathrm{end}_j(\beta)$. This check is sufficient because it requires both the number of processes in each user state to be the same at the beginning and end of the path and the state component of the configurations at the beginning and end of β to be the same. We check that the length of β is at least two by checking that transition-count(β) > 0.

[4] Recall that if $c = (x, \vec{v})$, then state(c) = x.

(b′) β is a cycle starting with a proper configuration

$$\leftrightarrow (\text{init-state-nbr}(\beta) = \text{end-state-nbr}(\beta))$$
$$\wedge(\text{init-state-nbr}(\beta) \text{ denotes a proper state})$$
$$\wedge \bigwedge_{j,\, j \in S_U} (\text{init}_j(\beta) = \text{end}_j(\beta)) \wedge \text{transition-count}(\beta) > 0.$$

(c′) a final configuration appears in $\beta \leftrightarrow \text{final-count}(\beta) > 0$.

In the following condition, for each state j, define internal_j to be true if there is an internal process transition defined from state j, and false, otherwise.

(d′) for all states $j \in S_C \cup S_U$, β is j-sepfair

$$\leftrightarrow \bigwedge_{j,\, j \in S_C \cup S_U} \Big[\text{exit}_j(\beta) > 0$$
$$\vee(\text{init}_j(\beta) = 0 \wedge \text{end}_j(\beta) = 0)$$
$$\vee\Big(\neg\text{internal}_j \wedge \Big(\bigwedge_{k,\, k \text{ enables } j}$$
$$(\text{init}_k(\beta) = 0 \wedge \text{exit}_k(\beta) = 0)\Big)\Big)\Big].$$

In order to express the above conditions as a reachability problem, we construct a new VASS G' from the VASS $G = \text{VS}(C, U, A_{\neg f})$. For each state s of G, let s^α, s^β, denote new states, distinct from each other. We call a state of the form s^α an *α-state*, and a state of the form s^β a *β-state*. The states of G' will be $\{s^\alpha, s^\beta \mid s \text{ is a state of } G\}$. The vectors in the configurations of G' will have all the components of G, plus the following new components which we refer to by name. For each $j \in S_C \cup S_U$, we have the new components init_j, end_j, enter_j, and exit_j. We also add components called transition-count and final-count.

Intuitively, the transitions of G' are defined so that G' simulates a path of G while also keeping track of the values of the functions in conditions (b′)–(d′). G' also nondeterministically guesses where to divide the path into segments α and β. Suppose $c_0, \ldots, c_j, \ldots, c_k$ is a path in G, where $c_i = (s_i, \vec{v}_i)$ for $i = 0, \ldots, k$. Then there is a path of G' of the form $(s_0^\alpha, \vec{v}_0'), \ldots, (s_j^\alpha, \vec{v}_j'), (s_j^\beta, \vec{v}_j'), \ldots, (s_k^\beta, \vec{v}_k')$, where the projection of $\vec{v}_i'$ onto the components of vectors in G is $\vec{v}_i$, for $i = 0, \ldots, k$. Intuitively, in this path, G' guesses that the path α is $(s_0, \vec{v}_0), \ldots, (s_j, \vec{v}_j)$, and the path β is $(s_j, \vec{v}_j), \ldots, (s_k, \vec{v}_k)$.

For each state s of G, there are transitions of G' in state s^α or s^β that update the original vector components of G in the same way that G updates them in state s. That is, if $(s, t, \vec{v})$ is a transition of G, then $(s^\alpha, t^\alpha, \vec{v}^\alpha)$ and $(s^\beta, t^\beta, \vec{v}^\beta)$ are transitions of G', where $\vec{v}^\alpha, \vec{v}^\beta$ update the original components of G in the same way $\vec{v}$ does. In addition, for each state s of G, G' has a transition from state s^α to state s^β, which does not change any of the vector components (i.e., the transition $(s^\alpha, s^\beta, \vec{0})$). Intuitively, when G' chooses such a transition, this marks the beginning of the segment β.

The VASS G' treats the new vector components as counters to compute the necessary functions. The counters are initially set to 0 by starting G' in a configuration with the vector $\vec{0}$. Reviewing conditions (b′)–(d′), the values to be computed are $\text{init}_j(\beta)$, $\text{end}_j(\beta)$, init-state-nbr(β), end-state-nbr(β), $\text{enter}_j(\beta)$, $\text{exit}_j(\beta)$, transition-count(β), and final-count(β). These values are equivalent, respectively, to values that can be directly counted, as follows:

$\text{init}_j(\beta) = \text{end}_j(\alpha)$. Compute $\text{end}_j(\alpha)$ by counting number of processes in process state j as long as G' is in an α-state. That is, each transition of G' increments or decrements init_j by the same amount that it changes the number of processes in state j. Stop counting when G' enters a β-state.

init-state-nbr(β) = end-state-nbr(α). This is also computed by making each component of G' increment or decrement this component appropriately.

end-state-nbr(β) = end-state-nbr($\alpha\beta$). Same as before, but is done by incrementing or decrementing this component over the entire path $\alpha\beta$.

$\text{end}_j(\beta) = \text{end}_j(\alpha \bullet \beta)$. Count number of processes in state j over the entire path.

$\text{enter}_j(\beta)$. Compute by counting number of transitions entering process state j, starting when G' enters a β-state.

The functions $\text{exit}_j(\beta)$, transition-count(β), and final-count(β) are computed in similar ways.

Thus, G' computes all of the functions of conditions (b′)–(d′) in its new vector components while it simulates a path of G. Let us say that a vector $\vec{v}$ of a configuration of G' satisfies (b′)–(d′) if the conditions are satisfied by the values of the new coordinates in the vector $\vec{v}$. We can now see that the original VASS G has a fair path starting from the initial configuration $(s_0, \vec{0})$ and containing infinitely many final configurations iff G' has a finite path starting from the configuration $(s_0^{\alpha}, \vec{0})$ and reaching a configuration whose vector satisfies conditions (b′)–(d′). Appendix C describes a general decidability result for VASS which shows that it is decidable whether G' has such a path. This completes the proof of Theorem 3.11. □

NOTE. In this proof of the decidability of the model-checking problem under fairness, the VASS that is constructed has many more coordinates than are strictly necessary It can be shown that the model-checking problem under fairness can be reduced to the reachability problem for a VASS of dimension only $4m + 5$, where m is the number of states of a user process [31]

THEOREM 3.15. *The model-checking problem for fair computations of the user process is decidable.*

PROOF. We prove the theorem by reducing this problem to the model-checking problem for fair computations of the control process. From the process definitions C and U, we construct process definitions C' and U', as shown in Figure 1. The basic idea is that U' can behave like either C or U, depending on the first action it takes. The symbols c, $\bar{c}$, u, and $\bar{u}$ are new communication symbols that are used by the control process to cause exactly one user process to act like the original control process, and cause all other users to act like the original user process. The process definition C' acts like the original user process after it starts some or all of the U' processes. The first two states reached by process C' are labelled with a new proposition P_{init}, which is

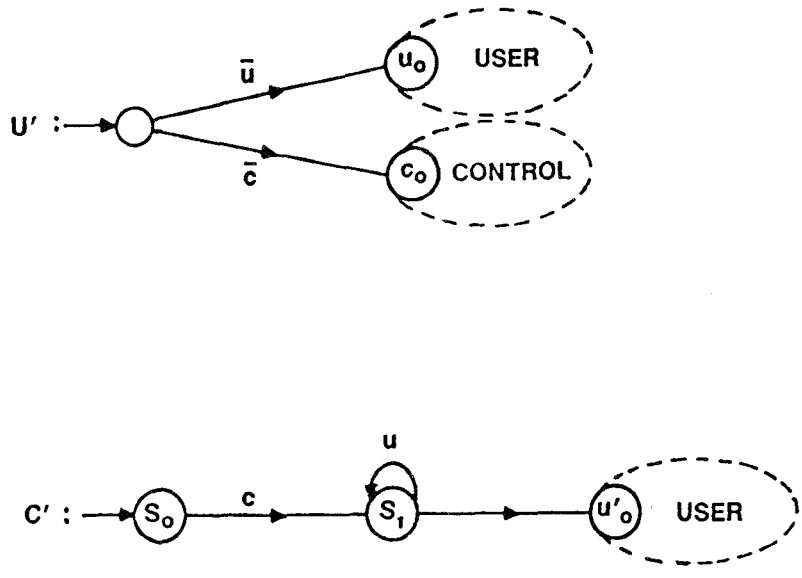


FIG. 1.

false in all other states. The following lemma shows how this construction can be used to solve the model-checking problem for the user process.

LEMMA 3.16. *The following are equivalent*:

(1) *For all n, all fair executions of a user process U in a system* $C \times U^n$ *satisfy f.*

(2) *For all n, all fair executions of* C' *in a system* $C' \times U'^n$ *satisfy* $P_{init}\ \mathbf{U}\ (\neg P_{init} \wedge f)$.

PROOF. As shown in Figure 1, a fair execution of the control process in a system $C' \times U'^n$ starts in the initial state s_0, enters and repeats state s_1 some finite number of times, and then continues with a computation that is equivalent to a fair execution of a user process in a system $C \times U^m$, for some $m \leq n$. Moreover, computations that are equivalent to each of the fair computations of the user processes in $C \times U^n$, prefixed by some states labeled with P_{init}, are included in the fair computations of the control process in $C' \times U'^n$. From this it is easily seen that (1) is equivalent to (2). □

Theorem 3.11 shows that the model-checking problem for fair computations of the control process is decidable. The following theorem indicates that the model-checking problem for fair computations is at least as hard as the reachability problem for VASSes. It has been shown in [17] that the reachability problem is EXSPACE-hard. It is generally believed that the reachability problem has much higher complexity. Indeed, the existing decision procedures [12, 19] have much higher complexity.

THEOREM 3.17. *If there is a deterministic algorithm for the model-checking problem for fair computations that is of time complexity* $f(n)$ *(space complexity* $(f(n))$*), then there exists a deterministic decision procedure for the reachability problem for VASSes that is of time complexity* $f(p(n))$ *(space complexity* $f(p(n))$*) where* $p(n)$ *in some polynomial in n.*

PROOF. We prove this theorem by presenting a polynomial time reduction from the reachability problem for VASSes to the problem of checking if there exists a fair execution of the control process in a system $C \times U^n$, for some $n \geq 0$, that satisfies a given PTL specification.

An algorithm for the reachability problem takes as input a VASS G and two configurations c and d, and decides if there is a path of G from c to d. There is a polynomial time-bounded transformation that takes G, c, and d as inputs and outputs another VASS, G^*, and two configurations $(s, \vec{0})$, $(t, \vec{0})$ of G^*, where G^* has the following properties:

(a) If $(q, q', \vec{a})$ is a transition of G^*, then there exists at most one component of $\vec{a}$ that is nonzero, and this component is $+1$ or -1.
(b) $(t, \vec{0})$ is reachable from $(s, \vec{0})$ in G^* iff d is reachable from c in G.

This transformation is straightforward.

Let G^*, s and t be as specified above. Now, we give the process definitions C, U, and a temporal specification f such that the following property is satisfied: There is an execution of the control process in the family $\{C \times U^n\}$ that satisfies f iff $(t, \vec{0})$ is reachable from $(s, \vec{0})$ in G^*.

Let the dimension of G^* be m, that is, each vector in the transitions of G^* is an m-vector of integers. The states of the control process include all the states of G^* together with two additional states t' and t''. The states of the user process are init, $u_1, \ldots, u_m$, and v. The state init is the initial state of a user process. Intuitively, if the configuration $(q, \vec{a})$ is reachable in G^*, then there exists a finite computation at the end of which the control process is in state q, and there are $\vec{a}[i]$ user processes in the user state u_i, for $1 \leq i \leq m$. We accomplish this as follows: Corresponding to every transition in G^*, say from state q to q' that increments the ith component of the configuration vector, we have the following pair of transitions labeled with complementary communication symbols. There is a transition in C from q to q', and there is one in U from the initial state to the state u_i. These two transitions allow the control process to change state from q to q' while at the same time incrementing the number of user processes in state u_i. Similarly, for every transition in G^* from state q to q' that decrements the ith component, we have the following pair of transitions labeled with complementary communication symbols. There is a transition in C from state q to q', and there is one in the user process from state u_i to the initial state.

The additional control states t' and t'' and the user state v are used to check for fairness. The idea is that in a fair computation the control process can reach state t' and remain there forever iff it can reach state t with no user process being in any of the states u_1 thru u_m. The former property can be asserted by a PTL formula. The detailed description of the processes is given below.

The action names used are ν, α_i, and γ_i, for $1 \leq i \leq m$. The initial state of U is init, and the initial state of C is s. The control and user process are shown in Figure 2.

For each i, $1 \leq i \leq m$, U has a transition from the initial state to state u_i, which is labeled with α_i, a transition from u_i to the initial state labeled with γ_i, and a transition from u_i to v labeled with ν. The control process definition C has the following transitions: If $(q, q', \vec{a})$ is a transition of G^* where for some

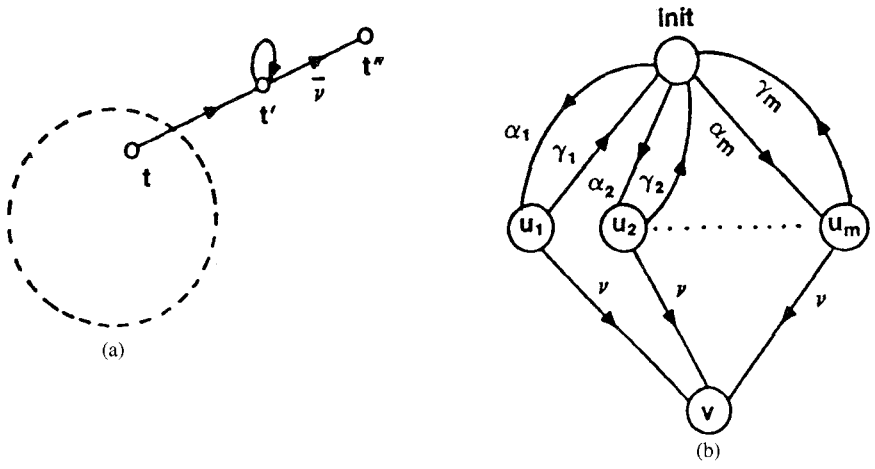


FIG. 2. (a) Control process. (b) User process.

i, $\vec{a}[i] \neq 0$ and for $j \neq i$, $\vec{a}[j] = 0$, then there is a transition in C from q to q'. If $\vec{a}[i] = 1$, then this transition is labeled with $\bar{\alpha}_i$; otherwise, it is labeled with $\bar{\gamma}_i$. There is an internal transition from t to t'. There is a self loop state t' that is an internal transition. There is a transition from t' to t'' that is labeled with $\bar{\nu}$. These are the only transitions in C. It is easy to prove that the configuration $(q, \vec{a})$ is reachable from $(s, \vec{0})$ in G^* iff there exists a finite computation sequence in $C \times U^n$, for some n, that ends in a global state with $\vec{a}[i]$ processes in the user state u_i, for $1 \leq i \leq m$. Let P be an atomic proposition which is true only in the state t' of the control process. Let f be the PTL formula $\mathbf{FG}(P)$.

CLAIM. *$(t, \vec{0})$ is reachable from $(s, \vec{0})$ in G^* iff there exists a fair computation such that the execution of C in this computation satisfies f.*

PROOF OF CLAIM. Assume that $(t, \vec{0})$ is reachable from $(s, \vec{0})$ in G^*. From our previous observation, it is easily seen that there is a finite computation sequence that starts in the initial state and reaches a global state where the control process is in state t and where all the user processes are in their initial state. Now, we can easily extend this computation sequence to an infinite fair computation in which all the user processes remain in the initial state, and the control process makes a transition to state t' and remains in this state forever. Clearly, the execution of the control process in this computation satisfies f. Assume that there is a fair computation such that the execution of the control process in this computation satisfies f. Clearly, in this computation, the control process reaches t' and remains in that state forever. Since this is a fair computation, it has to be the case that there is no user process in a state u_i, for $i > 0$, when the control process made a transition to t', otherwise due to the fairness condition the control process would have been forced to make a transition from t' to t''. From this, it follows that $(t, \vec{0})$ is reachable from $(s, \vec{0})$. This completes the proof of the claim.

It is straightforward to see that C, U, and f can be obtained by a polynomial time transformation. The lemma follows from the claim. □

3.4. DEADLOCKED COMPUTATIONS. So far, we have only considered infinite computations of a system. We have not included in the set of computations for model checking those computations that end in a deadlock state. Now, we extend the deadlocked computations to infinite sequences by repeating the final global state forever. If σ is a finite computation, we define the *extension* of σ to be the infinite sequence formed by repeating the last state of σ infinitely. If σ is an infinite computation, then we define its extension to be the same sequence σ. We say that an extension is deadlocked if it is the extension of a deadlocked (finite) computation. If $\mathcal{F}$ is a family of systems, then we define *Extensions*($\mathcal{F}$) to be the set of extensions of all computations of systems in $\mathcal{F}$. Similarly, we define *F-Extensions*($\mathcal{F}$), the set of fair extensions, to be the set of extensions of all fair computations of systems in $\mathcal{F}$. Recall that we defined all finite computations to be fair.

Now we consider the model-checking problem for Extensions($\mathcal{F}$) and FExtensions($\mathcal{F}$), for families $\mathcal{F}$ of the form $\{C \times U^n\}$. In addition to these problems, we also consider the *deadlock detection* problem, which is closely related. The deadlock detection problem is to determine if there is a finite computation of a system in a family $\{C \times U^n\}$. We show that all these problems are decidable, but are as hard as the reachability problem for a VASS.

As in Section 3.2, let $C = (S_C, R_C, I_C, \Phi_C)$, $U = (S_U, R_U, I_U, \Phi_U)$ be the definitions of the control process and a user process respectively. Let Control-Exten(C, U) and Control-FExten(C, U) be the projections of Extensions($\mathcal{F}$) and FExtensions($\mathcal{F}$), respectively, onto the control process. The model-checking problem for extensions for the control process consists of checking if every member of Control-Exten(C, U) satisfies a given PTL specification. The model-checking problem for extensions of fair computations and for projections onto a user process are defined similarly.

Let G be any VASS. We say that a configuration $(s, \vec{a})$ of G is a deadlock configuration iff there is no transition of G of the form $(s, s', \vec{e})$ such that $\vec{a} + \vec{e} \geq \vec{0}$. Roughly speaking, a configuration is a deadlock configuration if no transition of G is enabled in the configuration.

Theorem 3.18 given below has the following consequences. If there exists a deterministic algorithm for the deadlock detection problem or for the model-checking problem for extensions for the control process that is of time complexity $f(n)$ (space complexity $f(n)$), then there exists a deterministic algorithm for the reachability problem that is of time complexity $f(p(n))$ (space complexity $f(p(n))$) where $p(n)$ is a polynomial in n.

THEOREM 3.18. *There is a polynomial time reduction from the reachability problem for a VASS to the deadlock detection problem and to the problem of checking if there exists an execution e in Control-Exten(C, U) such that $\Phi_C(e)$ satisfies a given PTL specification.*

PROOF SKETCH. We use the same reduction as given in the proof of Theorem 3.17 with certain modifications. Let G^*, s, t, C, and U be as given in the proof of Theorem 3.17. We make the following modifications to C and U. The basic idea of the modifications is that in a fair extension the control process can reach and stay forever in state t' iff at the time it reaches t' the global state is a deadlock state. In the user process definition U, we delete the node v. The following modifications are made to the control process. The self loop in state t' is deleted, and self loops are introduced in all other states of the

control process. All these self loops are internal transitions of the control process. The internal transition from t' to t'' is replaced by the following transitions. For each i, such that $1 \leq i \leq m$, a transition labeled with $\bar{\gamma}_i$ is introduced from state t' to t''. The resulting process definitions C and U are shown in Figure 3. Let P be an atomic proposition that is true only in the state t' and f be the formula $\mathbf{FG}(P)$.

CLAIM. *The following are equivalent*:

(*a*) $(t, \vec{0})$ *is reachable from* $(s, \vec{0})$ *in* G^*.

(*b*) *There is an execution in Control-Exten*(C, U) *that satisfies* f.

(*c*) *There is a deadlocked computation in the family* $\{C \times U^n\}$.

PROOF. We show that (a) implies (b), (b) implies (c), and (c) implies (a). Assume (a), that is, $(t, \vec{0})$ is reachable from $(s, \vec{0})$ in G^*. From the arguments used in the proof of Theorem 3.17, it follows that there is a finite computation at the end of which the control process is in state t and all the user processes are in the initial state. In this global state, the control process can make a transition to t' and cause a deadlock. In the extension of this computation, the control process remains forever in state t', and thus, its execution satisfies f. Hence, (a) implies (b). Now we show that (b) implies (c). Assume (b); that is, there exists an execution in Control-Exten(C, U) that satisfies f. Clearly, in this execution, the control process reaches t' and remains in that state forever. It has to be that when the control process reaches state t', there is no user process in any of the user states u_1 through u_m; otherwise, the control process will be forced to make a transition to state t''. Hence, (c) holds. Now we show that (c) implies (a). Assume (c); that is, there is a deadlocked computation. Since all states of C except t' have self loops that are internal transitions, it has to be that in the deadlock state, the control process is in state t', and none of the user processes is any of the states u_1 through u_m. From this, it follows that the global state in which the control process is in state t, is reachable, and none of the user processes is any of the states u_1 through u_m. From this and our previous observations, it follows that the configuration $(t, 0)$ is reachable from $(s, \vec{0})$ in G^*. Hence, (a) holds. This completes the proof of the claim, and the theorem follows. □

THEOREM 3.19. *The deadlock detection problem is decidable.*

PROOF. Let Count(σ, s), where σ is a global state of a system and s is a state of a user or control process, be the number of processes in state s in the global state σ. Notice that if s is a state of a control process then count(σ, s) is either 0 or 1. Let us say that a pair of process states, (s_i, s_j), is an *enabling pair* if two processes in states s_i, s_j can make a transition by synchronizing with each other. A global state σ has an enabled transition iff it has a process that can make an internal move or a pair of processes in an enabling pair of states. Thus, a state σ is deadlocked iff conditions (1), (2), and (3) hold, where the conditions are

(1) $\bigwedge_{s_i:\ s_i \text{ has an internal transition}} (\text{count}(\sigma, s_i) = 0)$

(2) $\bigwedge_{(s_i, s_j) \text{ an enabling pair}} (\text{count}(\sigma, s_i) = 0 \vee \text{count}(\sigma, s_j) = 0)$

(3) $\bigwedge_{(s_i, s_i) \text{ an enabling pair}} (\text{count}(\sigma, s_i) = 0 \vee \text{count}(\sigma, s_i) = 1).$

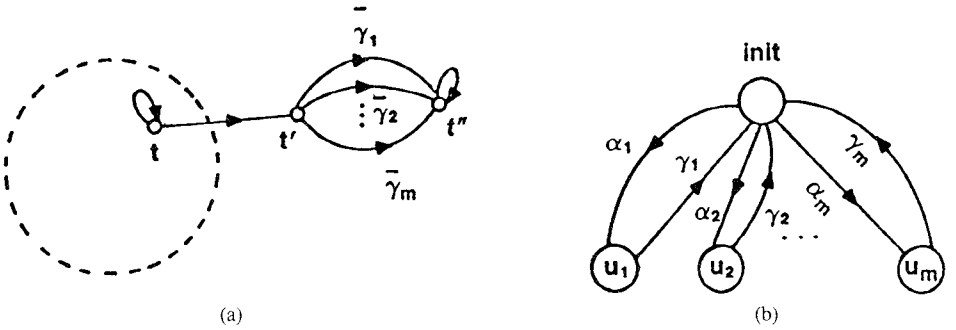


FIG. 3. (a) Control process. (b) User process.

Now we can reduce the deadlock detection problem to the reachability problem for VASSes. From the control and user processes C and U, we construct a VASS $G = \mathrm{VS}(C, U, A)$ where A is an automaton defined as follows: the automaton A has a single state which is its initial as well as final state; it remains in the same state on all inputs and accepts all inputs. Then, we extend G by adding a new coordinate for each control state. These new coordinates have the value 0 or 1, depending on which state the control process is in. Let G' be the resulting VASS.

We can describe the result of this construction as follows: First, for a global state σ, let $\mathrm{vec}(\sigma)$ be a vector of natural numbers such that for all states s of C or U, there is a coordinate of $\mathrm{vec}(\sigma)$ corresponding to s, and this coordinate has the value $\mathrm{count}(\sigma, s)$. Then for all global states σ, the following holds: σ is a state that can be reached in a computation of the family $\{C \times U^n\}$ iff the VASS G' can reach a configuration with the vector $\mathrm{vec}(\sigma)$.

Now we can use the result of Appendix C to show that the deadlock detection problem is decidable. The system of processes can reach a deadlocked state iff G' can reach a configuration satisfying a positive condition (in the terminology of Appendix C) on its coordinates. This shows that it is decidable whether a deadlocked state can be reached. □

THEOREM 3.20. *The model-checking problems for extensions and fair extensions for the control process are decidable.*

PROOF SKETCH. First, we consider the model-checking problem for extensions for the control process. Let f be the given PTL specification. We check if there exists an execution in Control-Exten(C, U) that satisfies $\neg f$. We split this problem into the following two cases: (i) there exists a nondeadlocked extension in which the execution of the control process satisfies $\neg f$; (ii) there exists a deadlocked extension in which the execution of the control process satisfies $\neg f$. We check for case (i) using the approach of Section 3.2.

We reduce case (ii) to the deadlock detection problem as follows. From C, $A_{\neg f}$, we obtain a new process definition C'' which roughly speaking is a cross product of C, $A_{\neg f}$. Let

$$A_{\neg f} = (Q, \Delta, \delta, I, F) \quad \text{and} \quad C = (S_C, R_C, I_C, \Phi_C).$$

Now,

$$C'' = (S'', R'', I'', \Phi'')$$

where

$$S'' = S_C \times Q,$$

$$R'' = \{((s, q), (s', q'), c): q' \in \delta(q, \Phi_C(s')) \text{ and } (s, s', c) \in R_C\},$$

$$I'' = \{(s, q): s \in I_C \text{ and for some } q_0 \in I,\ q \in \delta(q_0, \Phi_C(s))\},$$

and for any $(s, q) \in S''$, $\Phi''((s, q)) = \Phi_C(s)$. Recall that, for any $e = (e_0, e_1, \ldots)$ where $e_i \in S_C$ for all $i \geq 0$, $\Phi_C(e)$ denotes the sequence $(\Phi_C(e_0), \Phi_C(e_1), \ldots)$. We say that the state (s, q) of C'' is *compatible* if the automaton $A_{\neg f}$ when started in the state q, accepts the string $(\Phi_C(s))^\omega$, that is, the automaton has an accepting run on this string starting from the state q.

CLAIM. *The following are equivalent*:

(*a*) *There is a deadlocked extension of the system* $C \times U^n$, *such that the execution of the control process in this extension satisfies* $\neg f$.
(*b*) *There is a deadlocked computation of the system* $C'' \times U^n$, *such that the last state of* C'' *in this computation is a compatible state.*

PROOF OF CLAIM. Assume (a). Thus there is a deadlocked extension of the system $C \times U^n$ such that the execution of the control process in this extension satisfies $\neg f$. Consider the finite computation that produces this extension, and let $e = (e_0, \ldots, e_h)$ be the execution of the control process in this deadlocked computation. The sequence $e(e_h)^\omega$ is the execution of the control process in the deadlocked extension, and this sequence satisfies $\neg f$. Now, consider the sequence $\Phi_C(e)(\Phi_C(e_h))^\omega$. This string is accepted by $A_{\neg f}$. Let $(r_0, \ldots, r_{h+1}, \ldots)$ be the accepting run of $A_{\neg f}$ on this sequence, where r_{h+1} is the state of $A_{\neg f}$ after reading the first $h + 1$ symbols in the above input string. Clearly, the state (e_h, r_{h+1}) is a compatible state of C''. For all i, $0 \leq i \leq h$, let $b_i = (e_i, r_{i+1})$. From the definition of C'', it follows that b_i is a state of C''. In addition, from the way C'' is defined, it can easily be shown that there is a deadlocked computation in the system $C'' \times U^n$, such that $(b_0, \ldots, b_h)$ is the execution of C'' in this computation. Thus, (b) is true.

Now assume (b). That is, there exists a deadlocked computation in the system $C'' \times U^n$ such that the last state of the control process in this computation is a compatible state. Let $(b_0, \ldots, b_h)$ be the execution of the control process in this computation, where for all i, $0 \leq i \leq h$, $b_i = (e_i, r_i)$, e_i is a state of C and r_i is a state of $A_{\neg f}$. From the way we defined C'', the following can easily be seen. The sequence $(q, r_0, r_1, \ldots, r_h)$ is a run of $A_{\neg f}$ on the input string $(\Phi_C(e_0), \Phi_C(e_1), \ldots, \Phi_C(e_h))$, where q is an initial state of the automaton $A_{\neg f}$; there exists a deadlocked computation of the system $C \times U^n$ such that $(e_0, \ldots, e_h)$ is the execution of the control process in this computation. Since (e_h, r_h) is a compatible state of C'', it follows that $A_{\neg f}$ accepts the string $\Phi(e')$ where $e' = (e_0, \ldots, e_h)(e_h)^\omega$. Putting all the above observations together, we get (a). □

From the above claim, there exists an $n \geq 0$ such that (a) holds iff there exists an $n \geq 0$ such that (b) holds. The problem of checking if there exists an $n \geq 0$ such that (b) holds can be reduced to the deadlock detection problem as follows. For each state of C'' which is not a compatible state, we introduce a self loop which is an internal transition. Let C^* be the resulting process

definition. Now it should be obvious that there exists an $n \geq 0$ such that (b) holds iff there exists an $n \geq 0$ such that there exists a deadlocked computation in the system $C^* \times U^n$. Now using Theorem 3.19, we see that the model-checking problem for the extensions is decidable.

Now, consider the problem of model checking for the fair extensions. We check if there exists a fair extension such that the execution of C in this extension satisfies $\neg f$. Since every deadlocked extension is fair, we can split this problem into two cases: (i) there is an infinite fair extension such that the execution of C in this extension satisfies $\neg f$; (ii) there exists a deadlocked extension such that the execution of C in this extension satisfies $\neg f$. Checking for (i) is decidable by Theorem 3.11. Checking for (ii) can be done as explained above. □

Using the reductions of Sections 3.2 and 3.3, it is straightforward to show that both the model-checking problems for the extensions and fair extensions for the user process can be reduced to the corresponding problems for the control process. Thus, the problems for the user process are also decidable.

4. *Model Checking for Systems of Identical Processes*

In this section, we consider a restricted version of the model of processes considered in Section 3. Here, we consider systems consisting of an arbitrary number of processes with identical definitions. We present a polynomial time algorithm for the model-checking problem without fairness for this model of processes. Thus, the algorithm presented in this section is better than the one given in Section 3.2. The results of this section can be easily extended to systems consisting of families of processes where each family contains an arbitrary number of processes with identical definitions.

Let $U = (S, R, I, \Phi)$ be a process definition, and U^n denote a system of n processes having identical definitions given by U. Since in the system U^n all the processes have identical definitions, the sets of possible executions of these processes are identical. We define Exec(U) to be $\{t$: for some $n > 0$ t is the execution of a process in an infinite computation of the system $U^n\}$. For any $t \in S^\omega$, where $t = (t_0, t_1, \ldots)$ let $\Phi(t) = (\Phi(t_0), \Phi(t_1), \ldots)$. Recall that, for any $t \in S^\omega$, we say that t satisfies f iff $\Phi(t)$ satisfies f. For any $T \subseteq S^\omega$, let $\Phi(T)$ denote the set $\{\Phi(t) : t \in T\}$. Given U and a formula f, the *model-checking* problem is to determine if every member of Exec(U) satisfies f.

Let PTL$^-$ be the fragment of PTL that does not use the temporal modality **X**. In this section, we present an efficient model-checking algorithm that uses correctness specifications in PTL$^-$. First, we need the following definitions.

Let $t = (t_0 t_1, \ldots)$ be a finite or infinite sequence of elements drawn from a set of symbols Δ. A *segment* in t is a finite sequence $(t_i, t_{i+1}, \ldots, t_j)$ of identical elements such that t_j is distinct from t_{j+1}, and if $i > 0$, then t_{i-1} is distinct from t_i. Let $h(t)$ be the sequence obtained by replacing each segment in t by a single occurrence of the element in the segment. Note that if t is an infinite sequence and has a suffix that is a repetition of a single element, then this suffix will be retained in $h(t)$ because it is not a segment. We say that two sequences t and t' are *equivalent under stuttering* if $h(t) = h(t')$. Since we do not use the nexttime operator in PTL$^-$, sequences that are equivalent under stuttering satisfy the same PTL$^-$ formulas.

LEMMA 4.1. *For any $t, t' \in (2^{\mathcal{P}})^\omega$ such that t and t' are equivalent under stuttering and for any PTL$^-$ formula f, t satisfies f iff t' satisfies f.*

PROOF. By induction on the structure of the formula f. □

For technical convenience, we make the following two assumptions about U:

(i) There are no self loops in R, that is, there are no transitions of the form (s,s,c) in R;
(ii) For any pair of states (s, s') there is at most one transition of the form (s, s', c) in R.

If U does not meet the above assumptions, we can easily form a new process definition $U' = (S', R', I', \Phi')$ that meets the assumptions, and such that for any formula f, all the executions in Exec(U) satisfy f iff all the executions in Exec(U') satisfy f. The states of U' consist of the states of U together with some new states. The transitions of U' and the additional states of U' are as specified below. For each transition $t = (s, s', b)$ of U, U' has a new state q_t, and the transition (s, q_t, b) and an internal transition from q_t to s'. Φ' is the same as Φ on states in S; for the new states q_t, where $t = (s, s', b)$, $\Phi'(q_t)$ is defined to be the same set of atomic propositions as $\Phi(s')$. Clearly, the number of states in U' is $n + m$; the number of transitions in U' is $2m$ where n, m respectively are the number of states and number of transitions of U. U' can be obtained in time polynomial in size(U), and U' satisfies conditions (i) and (ii). Using Lemma 4.1, it is fairly straightforward to prove the following lemma.

LEMMA 4.2. *For any PTL$^-$ formula f, every element of Exec(U) satisfies f iff every member of Exec(U') satisfies f.*

In light of the above lemma, we assume henceforth that $U = (S, R, I, \Phi)$ satisfies conditions (i) and (ii). From here onwards, we fix the process definition U and fix the set EX = $\{t$: for some $t' \in$ Exec(U), t and t' are equivalent under stuttering$\}$. We call EX the *expanded* set of executions. The following lemma is a trivial consequence of Lemma 4.1.

LEMMA 4.3. *For a PTL$^-$ formula f, every member of Exec(U) satisfies f iff every member of EX satisfies f.*

Now, it is sufficient to check that every member of EX satisfies f. Our strategy is to reason about EX, the expanded set of executions of a process, without constructing a global-state graph. We construct a Buchi automaton A that accepts ϕ(EX). Then we check that there is no string that satisfies $\neg f$ and that is also accepted by A. The major steps of the model-checking algorithm are as follows:

(1) Determine the set of *reachable* states of a process, that is the set of all states s such that, for some $n > 0$, s appears in the execution of some process of the system U^n. Let this set of states be denoted by S'.
(2) Construct a directed graph $K = (S', R')$. EX, the expanded set of executions of a process is contained in the set of infinite paths of K starting from the initial states. However, every such path need not be an execution of a process.
(3) We show that there exists a set E of edges of K such that EX is exactly the set of infinite paths of K that start from an initial state and in which every edge not in E appears only a finite number of times. In this step, the edge set E is determined.
(4) Using K and E, construct the Buchi automaton A that accepts the set Φ(EX).

(5) Build the automaton $A_{\neg f}$ which accepts the set of strings that satisfy $\neg f$.
(6) In the final step, check that there is no string accepted both by A and $A_{\neg f}$.

We now describe the steps of the algorithm in more detail. In the first step, the following algorithm is used to compute S', the set of reachable states of a process: Initially S' is set to I, the set of initial states of U. Add a new state t to S' if the following condition is satisfied: Either there is a state $s \in S'$ such that $(s, t, \epsilon) \in R$, or there exist $s, s' \in S'$ such that for some $t' \in S$ and some $c \in \Sigma$, (s, t, c), $(s', t', \bar{c}) \in R$. The above step is repeated until no more new states can be added to S'. Clearly, at most $|S|$ iterations of this step are needed, and the above procedure can be implemented in time bounded by a polynomial in size(U). In the remainder of the section, let S' denote the value of the set S' after the termination of the above procedure. Recall that for any global state δ of the system U^n and for any p such that $1 \leq p \leq n$, let $\delta[p]$ denote the pth component of δ.

LEMMA 4.4. *Suppose* $S' = \{q_1, q_2, \ldots, q_m\}$.

(*a*) *A state s is in S′ iff for some* $n > 0$, *s appears in the execution of some process of the system* U^n;
(*b*) *For any positive integers* $n_1, \ldots, n_m$, *there is some* $k > 0$ *such that there is a finite computation sequence of* U^k *starting with an initial global state and reaching a state such that for each* i, $1 \leq i \leq m$, *there are at least* n_i *processes in the user state* q_i *in the last global state of the computation sequence.*

PROOF. We use the following observation in the proof: If σ, σ' are computation sequences of systems $U^n, U^{n'}$ respectively that start with initial global states and that end with global states δ, δ', respectively, then there exists a computation sequence σ'' of the system $U^{n+n'}$ that starts with an initial global state and ends with the global state δ'' where for $1 \leq p \leq n$, $\delta''[p] = \delta[p]$ and for $n < p \leq n + n'$, $\delta''[p] = \delta'[p - n]$. Now, we can prove (a) as follows:

(⇒) This direction is easily proved by induction on the number of iterations of the main step of the procedure that computes S' and by using the above observation.
(⇐) To prove this direction of (a), we can easily show that in any computation $\sigma_0, \sigma_1, \ldots$ of the system U^n the state of any process in the global state σ_i is in S'; this is accomplished by induction on i. Part (b) follows directly from the above observation. □

Now, we describe the second step of the algorithm, where we construct the graph $K = (S', R')$. The set $R' \subseteq S' \times S'$ contains exactly the following edges: For each state s in S', the self loop $(s, s) \in R'$; for $s, t \in S'$, if $(s, t, \epsilon) \in R$ then the edge (s, t) is in R'; for the states s, s', t, and t' in S', if (s, s', c), $(t, t', \bar{c}) \in R$ for some $c \in \Sigma$, then the edges (s, s'), (t, t') are in R'. Since we are not considering fairness, it is possible for a process to remain in a state forever. The self loops in R' model this possibility. They also model the possibility of stuttering in each state. A path in K is a sequence of states $(s_0, s_1, \ldots)$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R'$.

The following lemma relates the prefixes of executions of a process to the finite paths in K.

LEMMA 4.5. *The set of finite paths in K starting from an initial state is exactly the set of prefixes of elements of EX.*

PROOF. Consider a finite prefix α of an element of EX. It can be viewed as a prefix of an execution of a process with some states possibly repeated, that is, with stuttering. From our construction of K, it is easy to see that every state occurring in α is in S' and in addition, if s, s' appear consecutively in α then (s, s') is an edge in R'. From this it follows that α is a path in K. Again from the construction of K, it is straightforward to prove that every finite path in K is a prefix of an element in Ex. □

For an infinite sequence $\alpha \in S^\omega$, where $\alpha = (\alpha_0, \alpha_1, \ldots)$, let $\inf(\alpha)$ be the set of pairs of states (s, s') such that for infinitely many values of i, $\alpha_i = s$, $\alpha_{i+1} = s'$. Let ω-paths(K) be the set of infinite paths in K starting from an initial state. The following lemma states that the expanded set of executions EX is a subset of ω-paths(K). It can easily be proved using Lemma 4.5.

LEMMA 4.6. $Ex \subseteq \omega\text{-}paths(K)$.

There can be sequences in ω-paths(K) that are not in EX. Consider the process definition shown in Figure 4. States s_1 and s_5 are the start states. In this example, it is easily seen that there is a computation with $n + 2$ processes in which $(s_1 s_2)^n (s_3 s_4)^\omega$ is an execution of a process starting in the initial state s_1. The state graph corresponding to K is shown in Figure 5. Note that the path $(s_1 s_2)^\omega$ is not an execution of a process, and is not in EX because it is not equivalent under stuttering to any execution.

In step (3) of the algorithm, we show that there exists an edge set $E \subseteq R'$ such that $\text{EX} = \{\alpha : \alpha \in \omega\text{-paths}(K) \text{ and } \inf(\alpha) \subseteq E\}$.

Now we present the algorithm that determines E. This algorithm uses Linear Programming to determine if an edge e is in E. Roughly speaking, we use Linear Programming to determine if there is a finite computation sequence that starts and ends in the same global state, and uses the transition corresponding to the edge e at least once. If there is such a computation sequence, then it can be repeated infinitely to get a computation sequence that contains the transition corresponding to e infinitely many times.

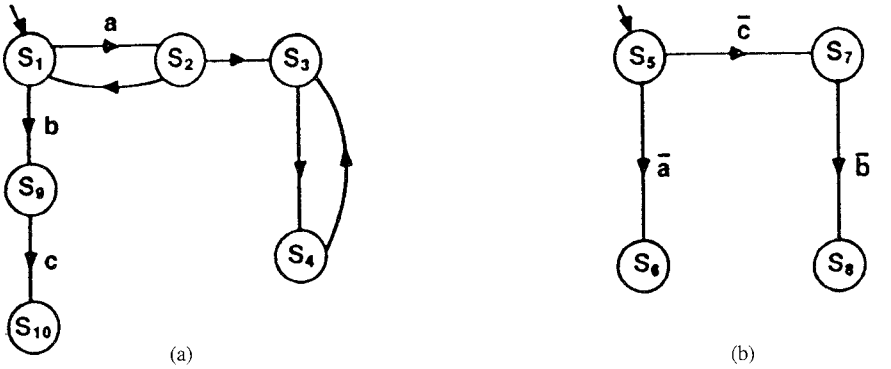
For each $e = (u, u') \in R'$, where e is not a self loop, we define label(e) to be the unique $c \in \Sigma$ such that $(u, u', c) \in R$; note that the uniqueness of label(e) is guaranteed by the assumptions (i), (ii) that we made about U. If label(e) is ϵ, then we say that e corresponds to an internal transition.

For any state $s \in S'$, let in(s) be the set of edges (s', s) such that $(s', s) \in R'$; let out(s) be the set of edges (s, s') such that $(s, s') \in R'$. For any $c \in \Sigma$, let edges(c) be the set of edges $e \in R'$ such that label$(e) = c$. For each edge $e \in R'$, let n_e be an integer. Intuitively, n_e will be the number of times the transition corresponding to e is used. We now set up a set of linear constraints that ensure the existence of a computation with the required property.

(A) For each state s,

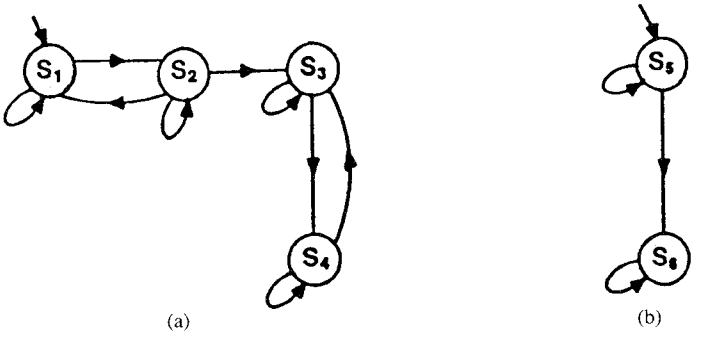
$$\sum_{e \in \text{in}(s)} n_e = \sum_{e \in \text{out}(s)} n_e.$$

These equations ensure that the total number of transitions entering a state is equal to the total number of transitions leaving the state.



(a) (b)

FIG. 4



(a) (b)

FIG. 5

(B) For each action $c \in \Sigma^+$,

$$\sum_{e \in \text{edges}(c)} n_e = \sum_{e \in \text{edges}(\bar{c})} n_e.$$

These equations ensure that the total number of transitions along edges labeled with c is equal to the total number of transitions along edges labeled with $\bar{c}$.

(C) For each edge e, $n_e \geq 0$.

For each edge e, we also make use of the single constraint

(D_e) $n_e > 0$.

Let E be the set of edges e such that the constraints (A)(B)(C)(D_e) have a (rational) solution. Since this is a homogeneous system of inequalities, it has an integer solution iff it has a rational solution. Using the polynomial time algorithm for Linear Programming [10], we can determine the set E in time polynomial in the size of K and hence in the size of U.

Next, we show in Theorem 4.8 that EX, the expanded set of executions of a process is exactly the set of infinite paths in K that start from an initial state and such that every edge that appears infinitely often is in E. We need the following technical lemma in the proof of Theorem 4.8.

LEMMA 4.7. *For any $s \in S'$, the self loop (s, s) is in E.*

PROOF. Let s be a state in S' and f be the edge (s, s). Let $n_f = 1$ and $n_e = 0$ for $e \neq f$. Clearly, these values satisfy the equations given by (A), (C). Since there are no self loops in the original process U, the above values also satisfy the equations given by (B). Thus, every self loop is in E. □

We use the following definitions in the proof of Theorem 4.8. Let $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_r)$ be a computation sequence of the system U^n, for some $n > 0$. Let p be an integer such that $1 \leq p \leq n$, and $s = (s_0, s_1, \ldots, s_r)$ be an execution of process p in σ. For any edge $e = (u, u') \in R'$, let $f(\sigma, e, p)$ be the number of values of i, such that $0 \leq i < r$, $s_i = u$ and $s_{i+1} = u'$. Intuitively, if e is not a self loop, then $f(\sigma, e, p)$ is the number of transitions taken by process p along e in the computation sequence σ. Now, define

$$g(\sigma, e) = \sum_{1 \leq p \leq n} f(\sigma, e, p).$$

Again, intuitively, if e is not a self loop then $g(\sigma, e)$ is the number of times a transition is taken along the edge e in σ by any process.

THEOREM 4.8. $EX = \{\alpha : \alpha \in \omega\text{-}paths(K)$ *and* $inf(\alpha) \subseteq E\}$.

PROOF. First, we prove that if $t \in \mathrm{EX}$ then $t \in \omega\text{-paths}(K)$ and $\inf(t) \subseteq E$. Let $t = (t_0, t_1, \ldots) \in \mathrm{EX}$. By Lemma 4.6, $t \in \omega\text{-paths}(K)$. Now, we prove that $\inf(t) \subseteq E$. By Lemma 4.7, if $e \in \inf(t)$ is a self loop, then $e \in E$. From the definition of EX, there is an execution $s \in \mathrm{Exec}(U)$ of a process that is equivalent to t under stuttering. Since s, t are equivalent under stuttering it is the case that, for any edge $e \in \inf(t)$, if e is not a self loop, then $e \in \inf(s)$. As a consequence, it is enough if we show that $\inf(s) \subseteq E$. Now consider a computation $\sigma = \sigma_0, \sigma_1, \ldots$ with n processes such that s is the execution of some process in σ. Let $E' = \{(q, q')$: for some i, $1 \leq i \leq n$, $(q, q') \in \inf(\delta_i)$ where δ_i is the execution of process i in the computation $\sigma\}$. Intuitively, for each $e = (q, q') \in E'$ which is not a self loop, a transition along e is taken infinitely often in the computation σ. For any i, j such that $j \geq i \geq 0$, let $\sigma(i, j)$ denote the computation sequence $(\sigma_i, \sigma_{i+1}, \ldots, \sigma_j)$. Clearly, certain global states appear infinitely often in the computation. Let i and j be integers such that $j > i$, $\sigma_i = \sigma_j$, for each $e \in E'$, $g(\sigma(i, j), e) > 0$, and for each $e \in R' - E'$, $g(\sigma(i, j), e) = 0$. Roughly speaking, i, j are integers such that a transition along every edge in E' is taken at least once between the instances i, j in σ and these are the only transitions taken between the instances i, j in σ. Clearly such i and j exist. Let $n_e = g(\sigma(i, j), e)$. Clearly, for $e \in E'$, $n_e > 0$, and for $e \notin E'$, $n_e = 0$. Since, for each state q in S' the number of processes in state q in the global state σ_i is same as the number of processes in state q in the global state σ_j, it is easily seen that equation (A) is satisfied for each state q in S'. It is also easy to see that the integers n_e satisfy the equations given by (B). Roughly speaking, this is due to the fact that every transition along an edge labeled with a communication symbol is synchronized with another transition taken along an edge labeled with a complementary communication symbol. From the above argument, it is clearly seen that $E' \subseteq E$. Hence, $\inf(s) \subseteq E$.

Now consider a sequence of states $\alpha = \alpha_0, \alpha_1, \ldots$ such that $\alpha \in \omega\text{-paths}(K)$ and $\inf(\alpha) \subseteq E$. Now we prove that $\alpha \in \mathrm{EX}$. We do this by constructing an

infinite computation in which one process has an execution that is equivalent under stuttering to α. Since the system of (A), (B), (C) is homogeneous, and since E is the set of edges e such that there is a solution for (A), (B), (C) with the constraint $n_e > 0$, we see that there is an integer solution for the system of inequalities given by (A), (B), (C) that simultaneously satisfies the constraints $n_e > 0$ for all $e \in E$. Let n_e, for each $e \in R'$, denote such a solution; clearly, for each $e \notin E$, $n_e = 0$. For each state $s \in S'$, let

$$N_s = \sum_{e \in \text{in}(s)} n_e.$$

For any global state δ and any state s, let $M_{\delta, s}$ be the number of processes in state s in the global state δ. Also, for any global state δ of the system U^n, and integer i such that $1 \leq i \leq n$, let $\delta[i := s]$ denote the global state in which the ith component has value s and all other components have same value as in δ. Now, we need the following claim. It states that starting from any global state δ in which there are at least N_s processes in state s and in which process 1 is in state q, and for any edge $(q, r) \in E$, we can have a computation sequence starting with δ and ending in a global state δ' such that for each user s, the number of processes in state s in δ' is same as that in δ and such that the only transition process 1 takes is along the edge (q, r).

CLAIM. *Let $e = (q, r) \in E$ where $q \neq r$ and δ be any global state such that $\delta[1] = q$, and for each user state s, $M_{\delta, s} \geq N_s$. Then, there exists a finite computation sequence β starting with δ, ending with some global state δ' and satisfying the following properties: for each state $s \in S$, $M_{\delta, s} = M_{\delta', s}$; the execution sequence of process 1 in β is qr^m for some $m > 0$.*

PROOF. We define the computation sequence by means of a procedure that generates it. The procedure uses a variable seq that after termination contains the desired computation sequence. First, let

$$n = \sum_{s \in S'} N_s.$$

The required procedure is given below. Initially, seq is set to δ.

While $\exists e' \in R'$ such that e' is not a self loop and $g(\text{seq}, e') < n_{e'}$ **Do**
(a) Choose an edge $e' = (q', r')$ and a process i as follows:
Let θ be the last global state in seq.
If this is the first iteration **then** choose e' to be e and i to be 1
else Choose e' so that e' is not a self loop and $g(\text{seq}, e') < n_{e'}$ and
choose i so that $i > 1$ and $\theta[i] = q'$.
Extend seq as follows:
(b1) **If** label$(e') = \epsilon$ **then**
extend seq by appending the global state$\theta[i := r']$ at the end.
(b2) **If** label$(e') = c \neq \epsilon$ **then**
choose and edge $e'' = (q'', r'')$ such that label$(e'') = \bar{c}$ and $g(\text{seq}, e'') < n_{e''}$;
Choose a process $j > 1$ and $j \neq i$ such that $\theta[j] = q''$;
Extend seq by adding the global state $\theta[i := r'][j := r'']$ at the end.
End while

The following are invariants of the **while** loop in the above procedure: (I1) For each $e \in R'$ that is not a self loop. $g(\text{seq}, e) \leq n_e$; (I2) For each $s \in S'$,

$$M_{\theta, s} \geq \sum_{h \in \text{out}(s)} \big(n_h - g(\text{seq}, h)\big);$$

(I3) For each $c \neq \epsilon$,

$$\sum_{\text{label}(f) = c} \big(n_f - g(\text{seq}, f)\big) = \sum_{\text{label}(f) = \bar{c}} \big(n_f - g(\text{seq}, f)\big).$$

From I1 and I2, it should be easy to see that we can choose some process i in step (a). From I3 and I2, it should be easy to see that we can choose an edge e'' and a process j in step (b2). Clearly, I1 is true initially. I2 holds initially, because, for any state s,

$$M_{\theta, s} \geq N_s = \sum_{h \in \text{in}(s)} n_h = \sum_{h \in \text{out}(s)} n_h.$$

I3 holds initially because the values n_f for $f \in R'$ satisfy the equations given by (B). It is fairly straightforward to show that I1, I2, I3 are invariants of the **while** loop.

Let β be the value of seq after the termination of the **while** loop and δ' be the last global state in β. Clearly, β is a computation sequence. Since in the first iteration, we choose e' to be e and p' to be 1, and, since process 1 is never chosen again, it is obvious that the execution sequence of process 1 in β is q^x for some $x > 0$. For any $f \in R'$, let $m_f = g(\beta, f)$. For $f \notin E$, $m_f = 0$. Now, consider any $f \in E$ such that f is not a self loop; we need to show that $m_f = n_f$. Since β is the value of seq after the termination of the **while** loop, it follows that $m_f \geq n_f$. Let β' be the value of seq when the edge f was last chosen in the **while** loop, that is, in step (a) or step (b2). Clearly $g(\beta', f) < n_f$ and β' is a prefix of β. Hence, it has to be that $m_f = n_f$. From this, we see that the values m_f for $f \in R'$ satisfy the equations given by (A) when we substitute m_f for n_f. From this, it follows that, for each s, $M_{\delta, s} = M_{\delta', s}$. □

In the path α, let i be the earliest instance with the property that every edge that is taken beyond the instance i in α is taken infinitely often, that is, $\forall j \geq i$ $(\alpha_j, \alpha_{j+1}) \in \text{inf}(\alpha)$. We can have a finite computation sequence γ that starts with an initial global state, ends with the global state π and that satisfies the following properties: for each state s, $M_{\pi, s} \geq N_s$ and the execution sequence of process 1 in γ is equivalent under stuttering to $(\alpha_0, \ldots, \alpha_i)$. Now, applying the previous claim repeatedly, it is easily seen that there exists an infinite computation sequence γ' starting with π such that the execution sequence of process 1 in γ' is equivalent under stuttering to $(\alpha_i, \alpha_{i+1}, \ldots)$. Now, consider the computation $\gamma \bullet \gamma'$; the execution of process 1 in this computation is equivalent under stuttering to α. Hence, $\alpha \in \text{EX}$. □

Step 4 of the algorithm uses K and E to construct a finite state Buchi automaton A that accepts exactly the set of strings given by $\Phi(\text{EX})$ and such that size(A) is linear in size(U). Construction of A is straightforward and is left to the reader.

Let f be the PTL$^-$ formula that is the correctness specification. In Step 5, the Buchi automaton $A_{\neg f}$ that accepts exactly the set of strings that satisfy $\neg f$

is constructed. Now to verify that every execution of a process satisfies f, it is enough to check that there is no string that is accepted by both A and $A_{\neg f}$. In Step 6, the techniques of [33] are used to check for this condition. This algorithm runs in time $O(\text{Size}(A) \cdot \text{Size}(A_{\neg f}))$. This completes the entire verification algorithm.

Let $n = \text{size}(U)$. Steps (1) and (2) can be implemented by a procedure that takes time bounded by a polynomial in n. In Step (3), we have to solve certain Linear Programming Problems. For this we use the polynomial time algorithm for Linear Programming [10]. From this it follows that steps (1), (2), and (3) can be implemented by a procedure that takes time $O(p(n))$ for certain polynomial $p(n)$. Steps (4) through (6) take time $O(\text{Size}(A) \cdot \text{Size}(A_{\neg f}))$. Since $\text{Size}(A)$ is proportional to n, and $\text{Size}(A_{\neg f})$ is proportional to $4^{|f|}$, it can be shown that the algorithm takes time $O(p(n) + n \cdot 4^{|f|})$ where p is a polynomial in n.

Example 4.9. Consider the process definition with five states, given by Figure 6, in which states s_1, s_4 are the initial states. Any process that starts in state s_1 acts as a consumer of messages generated by processes that start in the initial state s_4. Assume that we have atomic propositions S_i, for $1 \leq i \leq 5$, such that the only atomic proposition that is satisfied in state s_i is S_i. Also assume that whenever a process enters state s_3, it generates an output to the external world (not shown in the figure). Now, we would like to check that a process that starts in state s_1 generates only a finite number of external outputs. This property is asserted by the formula $S_1 \supset (\mathbf{FG}(S_3) \vee \mathbf{FG}(\neg S_3))$. This formula asserts that a process after some time either remains in state s_3 forever or remains outside s_3 forever; which is equivalent to the property that state s_3 is entered only a finite number of times. Using our algorithm we can automatically verify that the above property is satisfied by all executions of a process in a system of arbitrary number of processes with identical definitions given by Figure 6.

It is to be noted that the above property is a liveness property. Thus, although we do not consider fairness in our algorithm, some interesting liveness properties can also be verified by our algorithm, in addition to safety properties.

5. *Global Properties and Systems with Communication Networks*

In this section, we introduce some special rules of inference that allow us to use our decision procedures for certain problems that are not in the form considered thus far. Section 5.1 shows how the decision procedures can be used to verify that a system of processes maintains mutual exclusion. This is a global property of a system, not a property of the individual executions. Section 5.2 shows how to model certain systems having a form of process name variables or communication ports that can be dynamically assigned to different processes. In Section 5.3, we show how it is possible in some cases to use our decision procedures to reason about systems of processes having a communication network. We illustrate the methods with an example.

5.1. Model Checking for Mutual Exclusion. Consider a family $\mathcal{F} = \{C \times U^n\}$, and let the *critical region* be a subset of the process states of C and U. We would like to show that no computation of a system in $\mathcal{F}$ can reach a global state with more than one process in the critical region. We say that

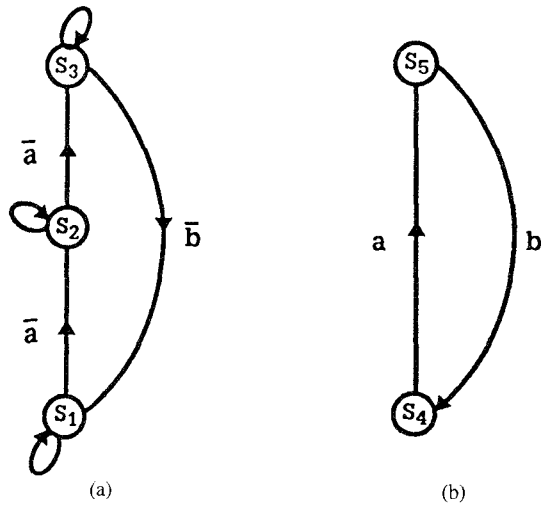


FIG. 6

computations with this property obey mutual exclusion, and similarly that systems (families) obey mutual exclusion if all their computations do.

By modifying the process definitions C and U, we can reduce this problem to reasoning about executions of single processes. First, we choose a pair of new communication symbols, c and $\bar{c}$, which do not appear in C or U. Taking the process definition C, for each state s in critical region, we add a new state e_s. From each state s in the critical region, we add transitions to state e_s labeled with the communication symbols c and $\bar{c}$. For each transition (s, s', u) of the process C from state s to a state s', we add a transition (e_s, s', u) from state e_s to s', with the same label. Let C' be the resulting process, as shown in Figure 7. If mutual exclusion is violated by a process in state s, the process can then enter state e_s. From this state, the process can continue its computation in the same way as the original process does in state s. A new process definition U' is formed similarly. Finally, we add a new atomic proposition E which is true in the added states e_s. The family $\{C' \times U'^n\}$ has the following useful property:

LEMMA 5.1. *The following are equivalent*:

(*a*) *All infinite* (*respectively, all infinite and deadlocked*) *computations of systems in* $\{C \times U^n\}$ *obey mutual exclusion.*

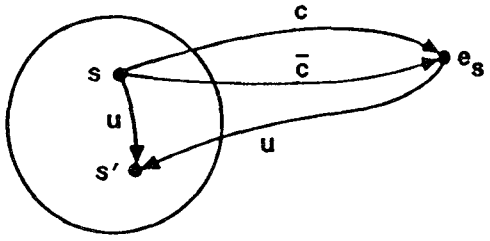
(*b*) *The formula* $\mathbf{G}(\neg E)$ *is satisfied by all the executions of the control and user processes in all infinite computations* (*respectively, in all extended computations*) *of the family* $\{C' \times U'^n\}$.

We can check for condition (b) by using the methods of Section 3.2. We can check for condition (b) for the case of extended computations by using the methods of Section 3.3.

COROLLARY 5.2. *It is decidable whether a family of the form* $\{C \times U^n\}$ *obeys mutual exclusion.*

The approach of Lemma 5.1 easily generalizes to verify the property that at any time, there are at most k processes in the critical region.

FIG. 7



5.2. SYSTEMS WITH COMMUNICATION PORTS. In our models of systems, when a process offers a certain communication symbol, it can communicate with any process that offers a complementary symbol. Certain applications require more structured patterns of communication. For example, one may want to model systems in which the control process communicates with one of the user processes, and then carries out a sequence of further communications with that particular user. We show that such a pattern of communication can be modeled in our first model of systems by introducing systems with *communication ports* that can be dynamically bound to different processes. Conceptually, in such systems, the control process has some fixed finite number of ports. At any time a port can be bound to a single user process and used to direct communication exclusively to that process. A port can then be switched to a different process.

The ports that we describe can also be thought of as a form of process name variables. A process name variable allows a process to store the identity of another process and to direct communications to the process named by the variable. The variable can later be set to name a different process. In effect, the construction in this section shows that we can model processes in an extended language having process name variables, where each process can have a fixed number of such variables.

Formally, we define the class of process definitions with p ports, where p is a fixed natural number, to be a subset of the process definitions given in Section 2, with certain restrictions that give the process more structure. The communication alphabet again consists of the special symbol ϵ and mutually disjoint sets of actions and complements of actions. The set of actions is divided into disjoint sets of *simple* actions and *port* actions. A simple action is a symbol with no further structure. A port action has one of the forms $\mathbf{connect}_i \cdot c$, $\mathbf{port}_i \cdot c$, or $\mathbf{disconnect}_i \cdot c$, where i is in $\{1, \ldots, p\}$ and c is a simple action. The complements of actions are simple complement actions such as $\bar{c}$, and port complement actions, which have the forms $\mathbf{connect}_i \cdot \bar{c}$, $\mathbf{port}_i \cdot \bar{c}$, or $\mathbf{disconnect}_i \cdot \bar{c}$, where $\bar{c}$ is a simple complement action. We say that simple actions and their complements are simple communication symbols; similarly, port actions and their complements are port communication symbols.

Intuitively, communication between the control process and a user with the symbols $\mathbf{connect}_i \cdot c$, $\mathbf{connect}_i \cdot \bar{c}$, respectively, binds the ith port of the control process to the particular user process. While this binding is in effect, the control process can communicate with the user by using symbols of the form $\mathbf{port}_i \cdot c$, $\mathbf{port}_i \cdot \bar{c}$. Finally, the port can be disconnected from the user by communication with symbols of the form $\mathbf{disconnect}_i \cdot c$, $\mathbf{disconnect}_i \cdot \bar{c}$.

To define the states of a process with p ports, we first choose a finite set S_b of *basic states*. The set of states of a process then has the form $S = \{s^{\mathscr{A}}, s \in S_b, \mathscr{A} \subseteq \{1, \ldots, p\}\}$. Intuitively, a state $s^{\mathscr{A}}$ is used to represent a process in a basic state s when it is connected to other processes on the ports in $\mathscr{A}$. The set $\mathscr{A}$ is called the active port set of the state $s^{\mathscr{A}}$. The set of initial states of a process with ports is a subset of $\{s^{\emptyset}, s \in S_b\}$.

The transitions of a process with ports are structured to preserve the meaning of the port symbols. Five kinds of transitions are allowed. In the following, $s, t \in S_b$, $\mathscr{A} \subseteq \{1, \ldots, p\}$, and c is a simple communication symbol

(1) $(s^{\mathscr{A}}, t^{\mathscr{A}}, \epsilon)$,
(2) $(s^{\mathscr{A}}, t^{\mathscr{A}}, c)$,
(3) $(s^{\mathscr{A}}, t^{\mathscr{A} \cup \{i\}}, \mathbf{connect}_i \cdot c)$, provided $i \notin \mathscr{A}$,
(4) $(s^{\mathscr{A}}, t^{\mathscr{A}}, \mathbf{port}_i \cdot c)$, provided $i \in \mathscr{A}$,
(5) $(s^{\mathscr{A}}, t^{\mathscr{A} - \{i\}}, \mathbf{disconnect}_i \cdot c)$, provided $i \in \mathscr{A}$.

We define the classes of user and control process definitions with ports to be subclasses of process definitions with ports. In a control process definition with ports, all transitions involving port communication symbols have port actions, that is, no part action complements appear. Similarly, in the transitions of a user process definition with ports, no port actions appear.

A family of systems with p ports has the form $\{C \times U^n\}$, where C (resp., U) is a control (resp., user) process definition with p ports. Intuitively, in the initial global states of any system in such a family, no ports are active. Transitions of types 1 and 2 can occur independently of the state of the ports. By using a pair of type-3 transitions, the control process can synchronize with a user on the symbols $\mathbf{connect}_i \cdot c, \mathbf{connect}_i \cdot \bar{c}$. After this, both processes enter states of the form $t^{\mathscr{A}}$ where $i \in \mathscr{A}$. In such states, type-4 and type-5 transitions are permitted. Observe that for each $i \subseteq \{1, \ldots, p\}$, at any time, no more than one user process can be in a state $s^{\mathscr{A}}$ with $i \in \mathscr{A}$. Also, in any reachable global state, the union of active port sets in the states of all user processes is equal to the active port set of the control process. Thus, the $\mathbf{port}_i \cdot c$ symbols provide exclusive communication between the control process and a single user. The $\mathbf{disconnect}_i \cdot c$ symbols reset the ith port to its initial state. This completes the description of processes with ports.

5.3. Model Checking for Systems with a Communication Network. The decision procedures we have presented in Sections 3 and 4 can also be used in some cases for verifying properties of distributed systems of almost identical finite-state CSP processes with a communication network, where the number of processes on the network is arbitrary. It is well known that most correctness problems for such systems are undecidable, because the systems have the power to simulate a Turing machine [1]. Thus, in this section, we present a method of reasoning that is sound but not complete.

To define such systems in our model of computation, we use an infinite set of communication symbols, with actions of the from $(i, j) : c$, where i and j are processes and c is a symbol in a message alphabet. We define the complement of $(i, j) : c$ to be $(j, i) : \bar{c}$. The rules for computation are unchanged; as before, any two processes that are in states having transitions on complementary communication symbols can synchronize. However, we now have enough symbols to describe a network. Typically, process i will use the symbol

$(i, j) : c$ to offer an action to process j, and process j will use the complementary symbol $(j, i) : \bar{c}$ to synchronize with P_i. If all the transitions in a system are of this form, then we call the system a *network system*.

As an example, consider the system for maintaining mutual exclusion on a ring network discussed in [4]. Initially, one process has a token that permits it to enter its critical region. The processes circulate the token around a ring network. In our model of computation, we can define the system of n processes, $S_n = (A_0, \ldots, A_{n-1})$, where the transitions of A_i are shown in Figure 8. Here, a_i is the index of the process definition A_i in the system. The subscripts $i + 1$ and $i - 1$ are taken mod n. (Note that to be consistent with our definition of a process, the actual values of $a_0, \ldots, a_{n-1}$ range from 1 to n.)

The critical region consists of the state C. The initial state of A_0 is C, while all other processes are defined to start out in state N, the noncritical region. A process in state N can enter its waiting state W by an ϵ-transition, Then, it waits to receive the token from the next lower process before it can enter its critical region. The state T is used by a process that has the token, but is not in its critical region. It permits the token to be circulated by processes without entering critical regions.

If S is a network system, so that all its transitions are labeled with process pairs, $(i, j):c$, then we define S^* to be the system formed by erasing the process pairs from all the transitions. That is, each communication symbol of the form $(i, j) : c$ is replaced by the symbol c. Now, it is clear that all computations of S are computations of S^*, but not vice versa. Thus, if we can prove that a property holds for all computations of S^*, then it holds for computations of S.

LEMMA 5.3. *Let $\mathcal{F}$ be a family of network systems and $\mathcal{F}^*$ be $\{S^* \mid S \in \mathcal{F}\}$. If a property holds for all computations of $\mathcal{F}^*$, then it holds for all computations of $\mathcal{F}$.*

Returning to the example, consider the family $\{S_n\}$, which consists of the ring network system of size n for all $n > 0$, and the family $\{S_n^*\}$, which is the set of all S_n^* for $n > 0$. It easy to see that when we erase process pairs from S_n, we are left with a system of the form $C \times U^n$. That is, the system S_n^* has the form $C \times U^n$, where C is the definition of the unique process that starts in the critical state, and U is the definition of the other processes.

Using the method of Section 5.1, we can use our decision procedures to verify that all systems in the family $\{S_n^*\}$ obey mutual exclusion. Hence, the family of network systems $\{S_n\}$ obeys mutual exclusion.

Although we believe the method of reasoning about S^* will be useful in some other cases for proving safety properties, it is not directly useful for reasoning about liveness properties, because for these properties, one usually considers only fair computations. It is not difficult to see that there can be a fair computation of a network system S that is not a fair computation of S^*. For instance, this can happen if a process is only enabled a finite number of times in the computation in S, but is enabled infinitely often in S^*. We are investigating other ways of using the model-checking algorithms to verify properties of network systems, but further development of this subject is beyond the scope of the present paper.

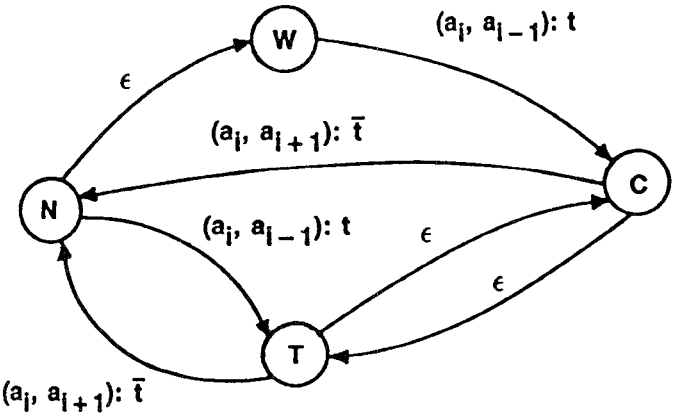


FIG. 8

6. *An Extended Logic*

So far, we have used the logic PTL to specify the properties of the executions of a single process. Now we introduce a new logic IPTL, which is an extension of PTL. This logic allows us to specify a wider class of properties. It uses the additional modality $\sqcap$, which is a universal quantifier over processes. We add the following rule for forming well-formed formulas. If f is an IPTL formula then $\sqcap(f)$ is also a formula in IPTL. We use the following abbreviation: $\sqcup \equiv \neg \sqcap \neg$.

A model of IPTL is a 4-tuple (I, S, Ψ, Φ) where I is a finite set of processes or indices, S is a set of states, $\Psi : I \to S^\omega$ associates with each process an ω-sequence of states, $\Phi : S \to 2^{\mathcal{P}}$. For $p \in I$, we denote $\Psi(p)$ by $s_p = (s_{p,0}, s_{p,1}, \ldots)$. An *interpretation* is a triple (M, p, i) where M is a model as given above, $p \in I$, and $i \geq 0$ is an integer. Intuitively, a model is a set of executions indexed by process names, and an interpretation specifies a model, a process name, and an instance on the execution of the process. We inductively define the relation $\models$, which specifies when an interpretation satisfies a formula:

$$(M, p, i) \models P, \quad \text{where } P \text{ is an atomic formula iff } P \in \Phi(s_{p,i});$$

$$(M, p, i) \models \sqcap(f) \quad \text{iff } \forall q \in I, (M, q, i) \models f;$$

$$(M, p, i) \models (f \mathbf{U} g) \quad \text{iff } \exists j \geq i \text{ such that } (M, p, j) \models g \text{ and } \forall k \text{ such that } i \leq k < j, (M, p, k) \models f;$$

$$(M, p, i) \models \mathbf{X} f \quad \text{iff } (M, p, i+1) \models f.$$

It is to be noted that, while $\sqcap$ is a universal quantification on processes, $\sqcup$ acts as an existential quantifier on processes. A formula is said to be *satisfiable* if there exists an interpretation that satisfies it, and a formula is said to be *valid* if all interpretations satisfy the formula. Using IPTL, we can express many interesting global properties of concurrent systems. The following formula expresses the property that, if a process requests a resource, eventually

some process is granted the resource

$$(\sqcup \textit{request} \supset \mathbf{F} \sqcup \textit{granted}).$$

In the above formula *request*, *granted* are atomic propositions.

IPTL is a special case of the multiprocess network logic of [25]; it is also a special case of the one-person logic of knowledge and time of [7] and [15]. In particular, IPTL is the same as the logic $KL_{(1)}$ of [7] interpretated over synchronous systems in which the processors do not forget and do not learn. The following theorem is independently observed by the authors and also in [7]. The reader is referred to [7] for the proof. The theorem shows that the validity problem for IPTL can be decided by an algorithm that is exponential space bounded and double exponential time bounded in the length of the formula.

THEOREM 6.1. *The set of valid formulas in IPTL is EXSPACE-complete.*

Now, we investigate the problem of checking if all the computations of a system consisting of a unique control process and an arbitrary number of user processes with identical definitions satisfy a given IPTL formula. Theorem 6.3 shows that this problem is undecidable. In this theorem, $\sum_1^0$ denotes the class of recursively enumerable sets, Π_1^0 denotes the class of complements of recursively enumerable sets.

First, we define what it means for a computation of a system of processes to satisfy an IPTL formula. To do this, consider a system of processes $(U_1, \ldots, U_n)$ where $U_1, \ldots, U_n$ are process definitions. For $1 \le i \le n$, let $U_i = (S_i, R_i, I_i, \Phi_i)$. We assume that the functions Φ_i are *consistent*, that is, for all i,j, $1 \le i,j \le n$ and for all s, if $s \in S_i \cap S_j$, then $\Phi_i(s) = \Phi_j(s)$. Corresponding to an infinite sequence of global states $\sigma = \sigma_0, \sigma_1, \ldots$ of the above system, we define a model M_σ of IPTL as follows: $M_\sigma = (J, S, \Psi, \Phi)$ where $J = \{1, \ldots, n\}$, $S = \bigcup_{0 \le i \le n} S_i$ and for any state $s \in S_i$, $1 \le i \le n$, $\Phi(s) = \Phi_i(s)$. Since, $\Phi_1, \ldots, \Phi_n$ are all consistent, it is the case that Φ is well defined. For any $i \in J$, $\Psi(i)$ is the projection of σ onto the ith coordinate. We say that the sequence σ *satisfies* an IPTL formula f iff for all $p \in J$, $(M_\sigma, p, 0) \models f$. It is easy to see that σ satisfies f iff σ satisfies $\sqcap f$.

LEMMA 6.2. *For a system of processes $(U_1, \ldots, U_n)$ and an IPTL formula, the problem of determining if all computations of the system satisfy f is decidable.*

PROOF SKETCH. Let W denote the system of processes $(U_1, \ldots, U_n)$ where $U_i = (S_i, R_i, I_i, \Phi_i)$, for $1 \le i \le n$. We prove the lemma by reducing the problem to that of checking if every path in a *Kripke structure* starting from some initial state satisfies a PTL formula. This problem is the standard model-checking problem for PTL and is decidable [16, 29]. First, we define a new set of atomic propositions $\mathscr{R} = \{P_i : P \in \mathscr{P} \text{ and } 1 \le i \le n\}$. Essentially, for each atomic proposition P in $\mathscr{P}$, $\mathscr{R}$ has n atomic propositions indexed by the process indices. Now, let $K = (T, R, \Theta, I)$ be a Kripke structure where T, R, Θ, I are given as follows: $T = S_1 \times \cdots \times S_n$, that is, T is the set of all global states of the system W; $R = \{(\sigma, \sigma') \in T \times T : \sigma'$ can be reached from σ in one computational step of the system $W\} : \Theta : T \to 2^{\mathscr{R}}$ such that for $\sigma = (\sigma_1, \ldots, \sigma_n)$, $\Theta(\sigma) = \{P_i : P \in \Phi_i(\sigma_i)$ and $1 \le i \le n\}$; $I = I_1 \times I_2 \times \cdots \times I_n$ is called the set of initial states. A path of K is a sequence

$\delta_0, \delta_1, \ldots$ such that for all $i \geq 0$, $(\delta_i, \delta_{i+1}) \in R$. It is clearly the case that every infinite path of K starting from an initial state is a computation of the system W.

Now, for any IPTL formula g that only uses atomic propositions from $\mathscr{P}$, we define a PTL formula $\rho(g)$, that uses atomic propositions from $\mathscr{R}$ so that for any infinite sequence δ of global states of the system W, δ satisfies g iff $\Theta(\delta)$ satisfies $\rho(g)$. First, for any IPTL formula g, we define a natural number $\mu(g)$, which is the *depth of nesting* of $\Box$ in g, inductively as follows: if g is an atomic proposition, that is, $g \in \mathscr{R}$, then $\mu(g) = 0$; if $g = g_1 \wedge g_2$, then $\mu(g) = \max\{\mu(g_1), \mu(g_2)\}$; if $g = \neg g_1$, then $\mu(g) = \mu(g_1)$; if $g = \Box g_1$ then $\mu(g) = 1 + \mu(g_1)$. Now, we define $\rho(g)$ by induction on $\mu(g)$. If $\mu(g) = 0$, then $\rho(g) = \wedge_{1 \leq i \leq n} g_i$ where g_i is obtained by replacing the occurrence of every atomic proposition $P \in \mathscr{P}$ by P_i. Now, assume that $\rho(g)$ is defined for all g such that $\mu(g) \leq k$, and let h be such that $\mu(h) = k + 1$. Then $\rho(h) = \wedge_{1 \leq i \leq n} h_i$ where h_i is obtained from h by the following operations: replace every atomic propositions P not in the scope of any $\Box$ by P_i; replace every subformula h' of the form $\Box h''$ that does not occur in the scope of any $\Box$ by $\rho(h'')$. The following claim is easily proved by straightforward induction on the depth of nesting of the modality $\Box$ and is left to the reader.

CLAIM. *Let $\delta = \delta_0, \delta_1, \ldots$ be any infinite sequence of global states of the system W. Then, for any IPTL formula f, δ satisfies f iff $\Theta(\delta)$ satisfies $\rho(f)$.*

Now, it follows that every computation of the system W satisfies f iff for every infinite path p of K starting from an initial state, $\Theta(p)$ satisfies $\rho(f)$. Now, the lemma follows from the results of [16] and [29]. □

As in Section 3, let $C \times U^n$ denote the system of processes consisting of a control process whose definition is given by C and n user processes having identical definitions given by U.

THEOREM 6.3. *The set of triples (C, U, f) that satisfy the following property is Π_1^0-complete: C and U are process definitions; f is an IPTL formula; for all $n \geq 0$, all the computations of the system $C \times U^n$ satisfy f.*

PROOF SKETCH. First, let Sat denote the set of all triples (C, U, f) such that for some $n \geq 0$, there exists a computation of the system $C \times U^n$ which satisfies f. We show that Sat is Σ_1^0-complete. From this result, it is easy to see that the set of triples (C, U, f) that satisfy the condition given in the statement of the theorem, is Π_1^0-complete.

First, we show that Sat is Σ_1^0-hard, by reducing the set of all encodings of *two-counter machines* [8] that accept an empty tape to Sat. A two-counter machine has a read-only input tape, a finite control, and two counters. Initially, both the counters are set to zero value. The finite control can increment or decrement each of the counters. It can also test for zero value for any of the counters.

Let M be a given two-counter machine. Now, we give a control process definition C, a user process definition U, and an IPTL formula f such that, for some $n \geq 0$, there is a computation of the system $C \times U^n$ that satisfies f iff

M accepts an empty input tape. We describe C and U informally. The process definition U is given in Figure 9. It has 3 states, u_0, u_1, and u_2, of which u_0 is the initial state. Roughly speaking, at any instance, the number of user processes in state u_i (for $i = 1, 2$) represents the value of counter i. The communication actions used are α_i and γ_i for $i = 1, 2$. The process definition C has the same states as the finite control of M. The initial state of C is the same as that of the finite control of M. Corresponding to each state of C or U there is an atomic proposition, with the same name as the state, which is true only in that state. The control process can increment or decrement the number of user processes in state u_1 or u_2 by offering appropriate communications. We represent the transitions of the finite control of M by a 4-tuple of the form $(s, s', *, *)$ or $(s, s', i, +)$ or $(s, s', i, -)$ or $(s, s', i, 0)$ or $(s, s', i, > 0)$ for $i = 1, 2$. All these transitions indicate a state change of the finite control of M from s to s'. The first transition does not refer to any counter, the next two transitions increment or decrement counter i, and the last two transitions indicate that the state change can occur only when counter i has zero value or has value greater than zero, respectively. We simulate the above transitions of M by the transitions (s, s', ϵ), $(s, s', \overline{\alpha}_i)$, $(s, s', \overline{\gamma}_i)$, (s, s', ϵ) of C in that order. Notice that corresponding to the transition $(s, s' i, 0)$ of M we only have an internal transition of C. We simulate the zero testing in the correctness specification, by asserting that the above transition of C should only be taken when there is no user process in the state u_i. First, we make the following assumptions about M: for each of the above type of transitions of M, $s \neq s'$; for every pair of states s and s' there is at most one transition of M that is in the above form (it is left to the reader to prove that for any given counter machine, by introducing new states, we can obtain another counter machine such that this machine accepts any empty tape iff the original machine accepts an empty tape; thus, there is no loss of generality due to these assumptions).

In order to give the correctness specification f, we define the following formulas. For every transition of M which is of the form $(s, s', i, 0)$, let $g_{ss'}$ be the formula $\mathbf{G} \sqcap ((s \wedge \mathbf{X} s') \supset (\sqcap \neg u_i))$. Note that $(s \wedge \mathbf{X} s')$ can be satisfied only by an execution of the control process. The formula $g_{ss'}$ asserts that there is no user process in state u_i when the transition from s to s' is taken by the control process. Similarly, for a transition of the form $(s, s', i, > 0)$ we can obtain a formula $g_{ss'}$ stating that the transition from s to s' occur only when there is at least one process in state u_i. Now, g is the conjunction of all $g_{ss'}$ such that there is a transition of M which is of the form $(s, s', i, 0)$ or of the form $(s, s', i, > 0)$. Let $f = \sqcup (s_0 \wedge g \wedge \mathbf{F} s_f)$ where s_0 and s_f are, respectively, the initial and final states of the finite control of M and where g is as specified above. It is straightforward to show that there is a computation of a system in the family $\{C \times U^n\}$ that satisfies f iff M accepts the empty input tape. From this, it follows that Sat is $\sum_1^0$-hard.

Now we prove that Sat is in $\sum_1^0$. Let C and U be the definitions of the control process and user process, respectively. Let f be an IPTL formula. Using the same approach as in [25], the following can easily be shown: For any $n \geq 0$, there is a computation of the system $C \times U^n$ that satisfies f iff there exists a computation of the system $C \times U^n$ that satisfies f and that is of the form $\alpha\beta^\omega$. Now, we give a nondeterministic algorithm that checks whether a triple (C, U, f) is in Sat. The algorithm guesses n, α, and β, and then checks that $\alpha\beta^\omega$ is a valid computation of the system $C \times U^n$ and verifies that this

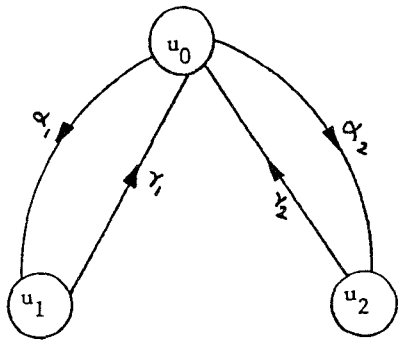


FIG. 9.

computation satisfies f using a procedure similar to that given in Theorem 2 of [25]. □

7. *Conclusion and Related Work*

In this paper, for the first time, we have considered the model-checking problem for two models of concurrent systems with an arbitrary number of finite-state processes. For these models, we have presented algorithms to check if the executions of a process satisfy a given PTL specification. We have also shown how process definitions with ports can be handled in the first model of processes. This allows us to handle the case where the control process can remember a bounded number of names of user processes. We have also shown how the algorithms can also be used to verify certain global properties such as mutual exclusion. We have illustrated the use of the algorithms by considering an example. We believe that other token-passing algorithms on rings and other networks, and some resource allocation problems, can be handled in this model at some level of abstraction.

The proof of the decidability result for the model-checking problem for fair computations for systems in the first model of processes reduces this problem to the problem of checking for absence of certain infinite paths in a Vector Addition System with States (VASS) and then shows that the later problem is reducible to the reachability problem for a VASS. The literature on Vector Addition Systems and Petri nets (see [22] for references) considers different notions of liveness and fairness. These notions are not related to the notion of fairness that we consider in this paper.

There has been much previous research on checking various properties of finite-state CCS/CSP processes. The complexities of checking for *lockouts*, absence of deadlocks and related properties in systems of finite-state processes have been analyzed in [9] and [14]. In these works, it is assumed that the number of processes is fixed.

The works that are closely related to ours are those in [3], [35], and [13]; the first of these was published more or less at the same time as the conference version of this paper [30]. In [3] the authors use a restricted version of an extended branching time temporal logic called ICTL* for specifying correctness properties. They present an approach for checking certain properties of the model of the processes consisting of a control process and an arbitrary number of user processes with identical definitions; their approach consists of obtaining another process U^*, called the *closure* of U, and showing that, for some

$r \geq 0$, the systems $C \times U^r \times U^*$ and $C \times U^{r+1} \times U^*$ are equivalent under a suitable notion of equivalence.

More recent works, such as [35] and [13], also consider families of systems of the form $\{C \times U^n\}$ and networks of processes of arbitrary size and use induction for proving correctness of these systems. In their approaches, the required correctness property is also specified as another process ϕ. The main feature of these works is to obtain another process I, called the *invariant process*, and prove the basis and induction steps by showing that $C \leq I$, $I \times U \leq I$, respectively; the correctness is proved by showing that $I \leq \phi$; here $\leq$ is a suitable partial order on processes. Once the invariant is obtained, then the other steps are accomplished using automated tools.

The main difficulty of the approaches mentioned above is that it requires a good deal of ingenuity to obtain the closure process or the invariant process. These methods are only partially automatic. None of these works prove any completeness results even for families of systems of the form $\{C \times U^n\}$. Using the results of [1], it can be shown that for the general case, that is, for networks of processes of arbitrary size, the invariant process I does not always exist.

We can extend the results of the paper for the following cases. Instead of PTL, we can also use finite-state Buchi automata on infinite strings to specify the properties of the executions of a single process. For the algorithm of Section 4, the specification automaton B should satisfy the following property: If B accepts a string t, then it should accept all the strings that are equivalent to t under stuttering. Since all our algorithms check for the absence of an execution that does not satisfy the given specification, we have to construct the complement B' of B, that is, we have to construct an automaton that accepts the set of strings that are not accepted by B. We modify the present algorithms by using B' in place of the automaton $A_{\neg f}$. Using the algorithm of [32], we can obtain B' in time $O(16^{|B|^2})$, and such that $|B'| \leq 16^{|B|^2}$. In this case, the algorithm given in Section 3 for the first model of processes without fairness will have time complexity double exponential in $|B|^2$, and the algorithm given in Section 4 for the second model will have time complexity $O(n \cdot 16^{|B|^2} + p(n))$.

We can extend the algorithms to the case when there is a lower bound k on the number of processes. For the first model of processes, it is straightforward to modify the VASS G to generate at least k user processes. It is also very straightforward to reason about families of the form $\{C \times U_1^{n1} \times \cdots \times U_m^{nm}\}$, with an arbitrary number of processes of type U_i for $i = 1, \ldots, m$, by combining the process definitions $U_1, \ldots, U_m$ into one process definition that makes a nondeterministic choice at its initial state, and then acts like one of the U_i. Since we are not considering fairness in the second model of processes, we do not have to modify the algorithm given in Section 4.

The algorithms we have given for the first model have high complexity. For this reason, the practical applicability of these results has to be further investigated. It may be possible to place sufficient realistic restrictions on systems and obtain more efficient model-checking algorithms for these problems.

Section 4 considers a model in which all processes have identical definitions and presents a polynomial time-bounded algorithm for checking the properties of executions of a process. This result also indicates that having a unique control process as in the first model makes verification a more difficult

problem. Although we have given a simple example, the result of the section seems to be mostly of theoretical interest at this point. Also, the algorithm of this section does not consider fairness. It is to be noted that the decidability of the model-checking problem for fair computations for this model follows by considering it as a special case of the first model. It will be interesting to investigate if we can do better than this. In particular, it will be interesting to investigate if there are algorithms for this problem whose time complexity is polynomial in the size of the process.

It will be interesting to consider other examples that can be verified using these algorithms. A more general question, for future research, is how to use the model-checking algorithms to reason about network systems, especially the liveness properties. It may be difficult to find completely automated algorithms. This suggests the approach of combining model checking with an axiom system to form an automatic proof checker for network systems.

Appendix A

In this appendix, we give a construction of a VASS denoted by the function VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$), which takes as arguments two process definitions $\mathscr{C} = (S_{\mathscr{C}}, R_{\mathscr{C}}, I_{\mathscr{C}}, \Phi_{\mathscr{C}})$, $\mathscr{U} = (S_{\mathscr{U}}, R_{\mathscr{U}}, I_{\mathscr{U}}, \Phi_{\mathscr{U}})$ and an automaton $\mathscr{A} = (Q, \Delta, \delta, J, F)$ where $\Delta = 2^{\mathscr{P}}$. Roughly speaking, certain infinite paths of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) simultaneously model the executions of the control process in systems of processes of the form $\mathscr{C} \times \mathscr{U}^n$ and the runs of the automaton $\mathscr{A}$ on these executions. Let $S_{\mathscr{U}} = \{u_1, \ldots, u_m\}$. VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) $= (V, E)$ where V is the set of nodes and E is the set of transitions. $V = (S_{\mathscr{C}} \times Q) \cup V_I \cup \{s_0\}$ where $S_{\mathscr{C}} \times Q$, V_1, $\{s_0\}$ are mutually disjoint; the elements in $S_{\mathscr{C}} \times Q$ are called *proper* states, those in V_I are called *intermediate* states, and s_0 is called the initial state. Each proper state consists of two components—a state of the control process and a state of the automaton $\mathscr{A}$. The second component is used to simulate the automaton $\mathscr{A}$ on the executions of the control process. The set of intermediate states is

$$V_I = \{(s, i, j), (s, i), (s, 0) : s \text{ is a proper state and } 1 \leq i, j \leq m\}.$$

We use the following notation in the remainder of the appendix. For any proper states s, t where $s = (e, q)$ and $t = (f, r)$, we say that t is a *successor* of s if $r \in \delta(q, \Phi_{\mathscr{C}}(f))$. For any $\tau, \tau' \in R_{\mathscr{C}} \cup R_{\mathscr{U}}$ where $\tau = (x, y, c)$ and $\tau' = (x', y', c')$, we say that τ, τ' are *complementary* process transitions if $c' = \bar{c}$. For any m-vector $\vec{v}$, we let $\vec{v}[i := x]$ denote the vector whose ith coordinate has value x and whose jth coordinate, for any $j \neq i$, has same value as that in $\vec{v}$. We let $\vec{0}$ denote the m-vector all of whose coordinates have value zero. For any i such that $1 \leq i \leq m$, let DEC1(i) denote the vector $\vec{0}[i := -1]$ and INC1(i) denote the vector $\vec{0}[i := 1]$. Similarly, for any i, j such that $1 \leq i, j \leq m$, let DEC2(i, j), INC2(i, j) denote m-vectors defined as follows: If $i = j$, then DEC2(i, j) $= \vec{0}[i := -2]$ and INC2(i, j) $= \vec{0}[i := 2]$; otherwise DEC2(i, j) $= \vec{0}[i := -1]$ $[j := -1]$ and INC2(i, j) $= \vec{0}[i = 1]$ $[j := 1]$.

A configuration $(s, \vec{v})$ of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) is called a *proper configuration* or an *intermediate configuration* if s is a proper state or is an intermediate state respectively. The configuration $(s_0, \vec{0})$ is called the *initial configuration*. Intuitively, each proper configuration of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) represents some global

states of a system $\mathscr{C} \times \mathscr{U}^n$ for some $n \geq 0$ and a state of the automaton $\mathscr{A}$. The proper configuration $c = (s, \vec{v})$ where $s = (e, q)$ represents a global state σ if the state of the control process in σ is e and there are $\vec{v}[i]$ processes in the user state u_i in σ.

We now discuss how the transitions of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) are defined. Each possible communication between processes and each internal transition of processes is modeled by some of transitions of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$). First, we consider the transitions that are used to model communication between the control process and a user process; the other cases are similar. Suppose $(e, f, c) \in R_c$ and $(u_i, u_j, \bar{c}) \in R_u$ are two process transitions. This pair of process transitions gives a possible communication between the control process in state e and a user process in state u_i. For every such pair of process transitions, we define a set of transitions of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$). Corresponding to the transitions $(e, f, c) \in R_c$ and $(u_i, u_j, \bar{c}) \in R_u$, and for every pair of proper states s, s' such that $s = (e, q)$, $s' = (f, r)$ for some q, r, and s' is a successor of s, VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) has a pair of transitions $t1 = (s, (s', j),$ DEC1(i)) and $t2 = ((s', j), s',$ INC1(j)). Intuitively, the transitions $t1$ and $t2$ work as follows: Suppose that the VASS is at a configuration $c = (s, \vec{v})$ where $s = (e, q)$. This configuration models the control process as being in state e and the automaton as being in state q. If $\vec{v}[i] > 0$, then c also models having a user process in state u_i. In this case, the VASS can use the transition $t1$ to reach the intermediate state (s', j). Note that the vector DEC(i) acts as a guard to test the condition $\vec{v}[i] > 0$ and to decrement the ith coordinate of the configuration vector. In state (s', j), the VASS can use the transition $t2$ to reach the proper state s' which is a successor of s. Note that the effect of the two vectors DEC1(i) and INC1(j) is to model the change in the state of a user process from state u_i to the state u_j. Communications between the user processes are modeled in a similar way.

The overall effect of the construction of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) provides the following correspondence: every computational step from a global state σ to a global state σ' is modeled by a path of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) of the form c, d, c' where c and c' are proper configurations that represent σ and σ', respectively, and d is an intermediate configuration. This path also models the change in state of the automaton corresponding to the change in state of the control process. Conversely, if c, d, c' is a path of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) where c is a proper configuration, and σ is a any global state that c represents, then there exists a global state σ' represented by c' such that σ' can be reached from σ in one computational step. From this it can be shown that paths of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) satisfy the following property. If $c = c_0, c_1, \ldots$ is in infinite sequence of proper configurations where, for all $i \geq 0$, $c_i = (s_i, \vec{v}_i)$, and $s_i = (e_i, q_i)$, then (1) holds iff (2) holds where (1) and (2) are given below. (1) There is a path π of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) starting from the initial configuration such that c is the sequence of all the proper configurations appearing in π. (2) The sequence $e = (e_0, e_1, \ldots)$ is an execution of the control process, and for some initial state r_0 of $\mathscr{A}$ $r_0, q_0, q_1, \ldots$ is a run of $\mathscr{A}$ on $\Phi_c(e)$. This property of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) can be proved using the same arguments as in the proof of Lemma 3.3.

It is to be noted that the transitions $t1, t2$, given in the previous paragraphs, can be replaced by the single transition $t = (s, s', \vec{a})$, where $\vec{a} =$ DEC1(i) + INC1(j), when $i \neq j$. However, when $i = j$, this causes

the following problem. Assume that $i = j$. In this case, $\vec{a} = \vec{0}$. Now if $c = (s, \vec{v})$ is a proper configuration, where $s = (e, q)$ and $\vec{v}[i] = 0$, then it is possible to reach the configuration $c' = (s', \vec{v})$ from c using the transition t. However, if σ is any global state that c represents, then the communication corresponding to the process transitions τ, τ' is not possible in σ as there is no process in state u_i in σ. As a result, we cannot use the single transition t in place of $t1, t2$ when $i = j$. For the sake of uniformity and ease of presentation, in all cases, we use pairs of transitions and intermediate states to model the computational steps. This does not cause any increase in the asymptotic complexity of the decision procedures.

In addition to the transitions that model the computational steps, VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) also has other transitions so that a proper configuration that represents an initial global state can be reached starting from the initial configuration. In order to define the set of transitions E, we need a subset, ST, of proper states defined as follows: $ST = \{(e, q) \in I_{\mathscr{C}} \times Q : q \in \delta(r, \Phi_{\mathscr{C}}(e))$ for some $r \in J\}$. Essentially, ST is the set of all pairs (e, q) such that e is an initial state of $\mathscr{C}$ and there is a transition of $\mathscr{A}$ from one of its initial states to state q on the input $\Phi_{\mathscr{C}}(e)$. Members of ST are the first proper states to occur on any path of VS($\mathscr{C}$, $\mathscr{U}$, $\mathscr{A}$) starting from the initial configuration.

The set of transitions $E = \cup_{1 \le i \le 6} E_i$ where the sets E_i, for $1 \le i \le 6$, are defined below. The transitions in E_1, E_2 are used to choose a proper configuration that represents an initial global state. The transitions in E_3, E_4 model communications between processes. The transitions in E_5, E_6 model internal moves of processes.

(i) $E_1 = \{(s_0, s_0, \vec{0}[i := 1]) : u_i \in I_{\mathscr{U}}\}$. Intuitively, for each state $u_i \in I_{\mathscr{U}}$, there is a transition in E_1 to generate an arbitrary number of user processes starting in the user state u_i.

(ii) $E_2 = \{(s_0, s, \vec{0}) : s \in ST\}$. These transitions model the move from s_0 to a state in ST.

(iii) The transitions in E_3 model the communication between the control process and a user process. Let $\tau = (e, f, c) \in R_{\mathscr{C}}$ and $\tau' \in (u_i, u_j, \bar{c}) \in R_{\mathscr{U}}$ be complementary process transitions. The transitions τ, τ' represent a possible communication between the control process and a user process. Let $s = (e, q)$ and $t = (f, r)$ be any proper states such that t is a successor of s. Corresponding to s, t, τ, τ' as given above, E_4 has two transitions denoted by $T1(s, t, \tau, \tau')$, $T1'(s, t, \tau, \tau')$ that are defined below: $T1(s, t, \tau, \tau') = (s, (t, j), DEC1(i))$ and $T1'(s, t, \tau, \tau') = ((t, j), t, INC1(j))$. It is to be noted that $T1(s, t, \tau, \tau')$, $T1'(s, t, \tau, \tau')$ correctly reflect the change in the state of the control process and they also change the number of user processes in a user state appropriately. The transition $T1(s, t, \tau, \tau')$ is called a *communication transition from* the pair of states (e, u_i), and $T1'(s, t, \tau, \tau')$ is called a *communication transition to* the pair of states (f, u_j). Formally, $E_4 = \{T1(s, t, \tau, \tau'), T1'(s, t, \tau, \tau') : s = (e, q), t = (f, r)$ are proper states such that t is a successor of s, and for some c, $\tau = (e, f, c) \in R_{\mathscr{C}}, \tau' \in R_{\mathscr{U}}$ are complementary process transitions$\}$.

(iv) The transitions in E_4 model synchronization between the user processes. For any proper states $s, t \in S_{\mathscr{C}} \times Q$ such that t is a successor of s and the control state component of s and t are identical and for any

complementary process transitions $\tau, \tau' \in R_{\mathscr{U}}$ where $\tau = (u_i, u_j, c)$, $\tau' = (u_p, u_q, \bar{c})$, E_4 has two transitions denoted by $T2(s, t, \tau, \tau')$, $T2'(s, t, \tau, \tau')$ that are defined below: $T2(s, t, \tau, \tau') = (s, (t, j, q), DEC2(i, p))$ and $T2'(s, t, \tau, \tau') = ((t, j, q), t, INC2(j, q))$. The transition $T2(s, t, \tau, \tau')$ is called a *communication transition from* the pair of states (u_i, u_p), and the transition $T2'(s, t, \tau, \tau')$ is called a *communication transition to* the pair of states (u_j, u_q). Formally, $E_4 = \{T2(s, t, \tau, \tau'), T2'(s, t, \tau, \tau') : s, t$ are proper states, and t is a successor of s, and the control state component of s and t are identical and τ, τ' are complementary process transitions in $R_{\mathscr{U}}\}$.

(v) The transitions in E_5 model the internal moves of the control process. For any $s, t \in S_{\mathscr{C}} \times Q$ such that $t = (f, r)$ is a successor of $s = (e, q)$ and $(e, f, \epsilon) \in R_{\mathscr{C}}$, E_5 has two transitions denoted by $T3(s, t)$ and $T3'(s, t)$ where $T3(s, t) = (s, (t, 0), \vec{0})$ and $T3'(s, t) = ((t, 0), t, \vec{0})$. The transition $T3(s, t)$ is called an *internal transition from state* e, and $T3'(s, t)$ is called an *internal transition to the state* f. Now, $E_5 = \{T3(s, t), T3'(s, t) : s, t$ are proper states such that t is a successor of s, and if $s = (e, q)$ and $t = (f, r)$, then $(e, f, \epsilon) \in R_{\mathscr{C}}\}$.

(vi) The transitions in E_6 model the internal moves of the user processes. For any proper states $s = (e, q)$ and $t = (e, r)$ where t is a successor of s and for any $\tau = (u_i, u_j, \epsilon) \in R_{\mathscr{U}}$, E_6 has two transitions denoted by $T4(s, t, \tau)$, $T4'(s, t, \tau')$ that are defined below: $T4(s, t, \tau) = (s, (t, j), DEC1(i))$ and $T4'(s, t, \tau) = ((t, j), t, INC1(j))$. Formally, $E_6 = \{T4(s, t, \tau), T4'(s, t, \tau') : s, t$ are proper states having same control-state component, and where t is a successor of s, and τ is an internal process transition in $R_{\mathscr{U}}\}$.

It is to be noted that there is only one transition frome each of the intermediate states. The transition from the intermediate state (s, i, j) (respectively, from the intermediate state (s, i)) leads to the proper state s and increments the i, j components (respectively, increments the ith component) of the configuration vector. The transition from the intermediate state $(s, 0)$ leads to the proper state s and does not alter the configuration vector.

It is also to be noted that for any transition $(s, s', \vec{a})$ of $VS(\mathscr{C}, \mathscr{U}, \mathscr{A})$, the absolute value of any coordinate of $\vec{a}$ is bounded by 2. It is straightforward to see that $|V| \leq |C| \cdot |\mathscr{A}| \cdot (m^2 + m + 2) + 1$.

Appendix B

In this appendix, we prove Lemma 3.5 using the results of [24] and [26]. We cannot directly use the results of these papers, because what we need is slightly different from what has been proved there. We reprove some of the results with the modifications. Consider a VASS G. Let the number of states of G be p, the number of transitions be n, and the dimension of G be m, that is, each vector in a transition of G be an m-vector. We also assume that for each transition $(s, s', \vec{a})$ of G, the absolute value of each component of $\vec{a}$ is bounded by L. A *u-configuration* (unconstrained configuration) is a pair $(s, \vec{a})$ where s is a state of G and $\vec{a} \in Z^m$. Notice that in a u-configuration the components of the vector $\vec{a}$ can be negative. An *unconstrained path* or *u-path* is a sequence of u-configurations $c_0, \ldots, c_i$ where for all $j < i$, c_{j+1} can be reached from c_j by a transition of G. Let F be a set of designated states of

G called the *final* states. For any u-configuration $c = (s, \vec{a})$, we let state(c), vec(c) respectively denote the state s and the vector $\vec{a}$. As in Section 3, for any transition $t = (s, s', \vec{a})$, we let source(t), target(t), and vec(t) respectively denote the states s and s' and the vector $\vec{a}$.

We use the following definitions more or less taken from [24] and [26]. For $0 \leq i \leq m$, we say that a vector $\vec{a}$ is *i-bounded* if $\vec{a}[j] \geq 0$ for all j such that $1 \leq j \leq i$, that is, all the first i components are nonnegative. Notice that by definition every vector is 0-bounded. If r is any positive integer, then $\vec{a}$ is *$i \sim r$ bounded* if it is i-bounded and in addition $\vec{a}[j] \leq r$ for $1 \leq j \leq i$, that is, all the first i components are nonnegative and are bounded by r. Let $c_0, \ldots, c_k$ be a u-path of G where $c_j = (s_j, \vec{a}_j)$ for $0 \leq j \leq k$. We say that the above u-path is i-bounded if for all j such that $0 \leq j \leq k$, $\vec{a}_j$ is i-bounded. It is $i \sim r$ bounded if $\vec{a}_j$ is $i \sim r$ bounded for all j such that $0 \leq j \leq k$. Notice that by definition, every u-path is 0-bounded. For two m-vectors $\vec{a}, \vec{g}$, we say that $\vec{a} \leq \vec{g}$ if for all i such that $1 \leq i \leq m$ $\vec{a}[i] \leq \vec{g}[i]$. We say that $\vec{a} < \vec{g}$, if $\vec{a} \leq \vec{g}$ and in addition for some i, $\vec{a}[i] < \vec{g}[i]$. Let $c = (s, \vec{a})$, $d = (s', \vec{g})$ be two u-configurations. We say that $c \leq d$ if $s = s'$ and $\vec{a} \leq \vec{g}$. We say that $c < d$ if $c \leq d$ and in addition $\vec{a} < \vec{g}$. A u-path $c_0, \ldots, c_k$ of G is said to be a *self-covering u-path* if there exists an $i < k$ such that $c_i \leq c_k$. It is said to be a *self-covering u-path with a final state* if there exist i and j such that $i \leq j \leq k$, $i < k$, $c_i \leq c_k$, and c_j is a final configuration, that is, state$(c_j) \in F$. Let c be a u-configuration. For $0 \leq i \leq m$, define $f(i, c)$ as follows: If there exists an i-bounded self-covering u-path with a final state starting from c, then $f(i, c)$ is the length of the shortest such u-path; otherwise, $f(i, c)$ is 0. Let $g(i) = \max\{f(i, c) : c \text{ is a } u\text{-configuration of } G\}$. We prove that $g(i)$ exists and is bounded. In [24] and [26], only self-covering paths are considered, whereas we are interested in self-covering paths with a final state. For this reason, we cannot directly use the results of [24] and [26], and we have to reprove them for the case of a self-covering path with a final state. The lemmas and proofs we present here are very similar to the lemmas and proofs given in [26].

LEMMA B1. *$g(0) = O(\{2L \cdot (m + n + p)\}^{d \cdot m})$ for some constant d.*

PROOF. We show that given any 0-bounded self-covering u-path with a final state starting from a u-configuration c_0, we can get another such u-path of length $O(\{2L \cdot (m + n + p)\}^{d \cdot m})$. Let $c_0, \ldots, c_i, \ldots, c_{i+k}$ be a 0-bounded self-covering u-path with a final state where $c_i \leq c_{i+k}$. We can assume that $i \leq p$ for the following reason (remember that p is the number of states in G). If $i > p$, then there have to be two integers v, w such that $0 < v < w$ and state(c_v) = state(c_w), and in this case, we can take the u-path $c_0, c_1, \ldots, c_v$ and from c_v onwards apply the sequence of transitions that were used in the u-path from c_w to c_{i+k} and obtain a shorter 0-bounded self-covering u-path.

By doing the above reduction repeatedly, we can get a 0-bounded self-covering u-path with a final state that satisfies the above property. Hence, we assume without loss of generality that $i \leq p$. Now, it is enough if we show that there is a u-path $e_0, \ldots, e_h$ of length $O(\{2L \cdot (m + n + p)\}^{d \cdot m})$ such that $e_0 = c_i$, $e_0 \leq e_h$ and some final configuration appears in this u-path. We prove this as follows:

Let S be the set of states of G appearing in some u-configuration from c_i to c_{i+k}, that is, $S = \{\text{state}(c_j) : i \leq j \leq (i + k)\}$. Note that S contains at least

one final state. For any transition t of G, let q_t be the number of times the transition t is taken from c_i to c_{i+k} in the above u-path. The variables q_t satisfy the following inequalities.

These inequalities are a consequence of the fact that $\text{vec}(c_i) \leq \text{vec}(c_{i+k})$. For each j such that $1 \leq j \leq m$,

$$\left(\text{sum of } \text{vec}(t)[j] \cdot q_t \text{ over all transitions } t \text{ of } G\right) \geq 0.$$

The following equations hold due to the fact the $\text{state}(c_i) = \text{state}(c_{i+k})$. They assert that for any state $s \in S$, the number of times state s is entered is equal to the number of times state s is exited and that state s is entered at least once. For each state $s \in S$,

$$\begin{aligned}&\left(\text{sum of all } q_t \text{ such that } \text{source}(t) = s\right)\\ &\quad - \left(\text{sum of all } q_t \text{ such that } \text{target}(t) = s\right) = 0;\\ &\quad \left(\text{sum of all } q_t \text{ such that } \text{target}(t) = s\right) \geq 1.\end{aligned}$$

The following equations assert that every state not in S is never entered or exited. For each state $s \notin S$,

$$\begin{aligned}\left(\text{sum of all } q_t \text{ such that } \text{source}(t) = s\right) &= 0;\\ \left(\text{sum of all } q_t \text{ such that } \text{target}(t) = s\right) &= 0.\end{aligned}$$

Now consider any positive integer solution for the above set of inequalities, and let $e_0 = c_i$. Clearly, $\text{state}(e_o) \in S$. It should be easy to construct a u-path $e_0, \ldots, e_h$ such that each transition t is taken exactly q_t number of times and such that $e_0 \leq e_h$. It should be clear that for every state $s \in S$, there exists $j \leq h$ such that $s = \text{state}(e_j)$. Hence, a final u-configuration appears in the above u-path. Since p, m, and n, respectively, are the number of states in G, the dimension of G, and the number of transitions in G, it follows that the number of inequalities in the above system is at most $2(p + m)$, the number of variables is n, and the maximum absolute value of any constant is L. Theorem 13.4 and the associated corollary in [21] states that if a system consisting of u inequalities in v number of variables has a positive integer solution, then it has positive integer solution in which the value of any variable is bounded by $(u + v) \cdot (1 + c_{\max}) \cdot (u \cdot c_{\max})^{2u+3}$, where $c_{\max}$ is the maximum absolute value of any constant appearing in the inequalities. Using this result, we see that there exists a solution for our system of inequalities in which the value of

$$q_t = O\left(L \cdot (n + m + p) \cdot \{2L \cdot (m + p)\}^{c \cdot 2(m+p)+3}\right)$$

for some constant c. Clearly, $g(0) \leq p +$ sum of all q_t. Using this and the fact that $2c \cdot (m + p) + 3 \leq d \cdot (m + p)$ where d is an appropriate constant, we see that

$$g(0) = O\left(n \cdot L \cdot (n + m + p) \cdot \{2L \cdot (m + p)\}^{d \cdot (m+p)}\right).$$

Now, it should be clear that

$$g(0) = O\left(\{2L \cdot (m + n + p)\}^{d \cdot (m+p)}\right),$$

where d is some other constant. □

The following lemma can be proved exactly on the same lines of Lemma 2.2 of [26], and hence the proof is omitted.

LEMMA B2. *If there exists an $i \sim r$ bounded self-covering u-path with a final state starting from the u-configuration c then there exists such a u-path starting from c and is of length $< (2 \cdot L \cdot p \cdot r)^{m^k}$ for some constant k.*

The following lemma is also proved exactly on the same lines of Lemma 2.4 of [26].

LEMMA B3. $g(i + 1) < (2 \cdot L^2 \cdot p \cdot g(i))^{m^d}$ *for some constant d.*

PROOF. Consider an $(i + 1)$-bounded u-configuration c_0 such that there is an $(i + 1)$-bounded self-covering u-path with a final state $c = c_0, \ldots, c_i, \ldots, c_{i+k}$ such that $c_i \leq c_{i+k}$. We want to show that we can get a shorter such path. The proof is split into the following two cases.

Case 1. The u-path c is $(i + 1) \sim (L \cdot g(i))$ bounded. In this case, using Lemma B2 with $r = L \cdot g(i)$, we can get an $(i + 1)$-bounded self-covering u-path with a final state starting from c_0 and which is of length $O((2 \cdot L^2 \cdot p \cdot g(i))^{m^d})$.

Case 2. Case 1 does not hold. Let j be the smallest integer such that for some $q \leq (i + 1)$, $\text{vec}(c_j)[q] > L \cdot g(i)$, that is, the qth coordinate of $\text{vec}(c_j)$ is greater than $L \cdot g(i)$. Without loss of generality, we can assume that $q = i + 1$. Now consider the u-path $c_0, c_1, \ldots, c_{j-1}, c_j$. For all x, y such that $0 \leq x < j$, $1 \leq y \leq i + 1$, $\text{vec}(c_x)[y] \leq L \cdot g(i)$. Now, if v, w are integers such that $0 \leq v < w < j$ and $\text{state}(c_v) = \text{state}(c_w)$ and $\text{vec}(c_v)$, $\text{vec}(c_w)$ agree on the first $i + 1$ coordinates, then we take the u-path $c_0, c_1, \ldots, c_v$ and extend it by applying the sequence of transitions used in the u-path $c_w, c_{w+1}, \ldots, c_j$ and obtain a shorter path that maintains all the required properties. By repeatedly doing the above reduction, we can obtain a u-path $e = e_0, e_1, \ldots, e_v$ of length $\leq p \cdot (L \cdot g(i))^{i+1}$ such that $e_0 = c_0$, all the u-configurations excepting e_v are $(i + 1) \sim (L \cdot g(i))$ bounded, and such that $\text{state}(e_v) = \text{state}(c_j)$, $\text{vec}(e_v)$ and $\text{vec}(c_j)$ agree on the first $i + 1$ components. Since $\text{vec}(c_j)[i + 1] > L \cdot g(i)$, it is also the case that $\text{vec}(e_v)[i + 1] > L \cdot g(i)$. Now, for all j such that $0 \leq j \leq (i + k)$, there exists an i-bounded self-covering u-path with a final state starting from c_j. (This can be seen as follows: For any j such that $0 \leq j \leq i$, the suffix of c starting from c_j gives us such a u-path. For any j such that $i < j \leq (i + k)$, we can obtain an i-bounded self-covering u-path with a final state starting from c_j by taking the suffix of c starting from c_j and extending it by using the sequence of transitions taken from c_i to c_{i+k}.) From this, it follows that there exists an i-bounded self-covering u-path with a final state starting from e_v, and hence there exists such a u-path e' starting from e_v which is of length $\leq g(i)$. Clearly, for every u-configuration f in e', the value of the $(i + 1)$st coordinate in $\text{vec}(f)$ never goes below $-L \cdot g(i)$. Now consider the u-path ee' (i.e., the u-path e

followed by the u-path e'). It should be clear that this is an $(i+1)$-bounded self-covering u-path with a final state and is of length $\leq p \cdot (L \cdot g(i))^{i+1} + g(i)$ and hence is less than $(2 \cdot L^2 \cdot p \cdot g(i))^{m^d}$ for some constant d. □

Now applying Lemma B3 repeatedly, we can show that

$$g(m) = O\left((2 \cdot L^2 \cdot p)^{m^{k \cdot m}} \cdot g(0)^{m^{k \cdot m}}\right)$$

for some constant k. Using Lemma B1 and substituting for $g(0)$, we get

$$g(m) = O\left((2 \cdot L^2 \cdot p)^{m^{c \cdot m}} \cdot \{2L \cdot (m+n+p)\}^{c \cdot (p+m) \cdot m^{c \cdot m}}\right)$$

for some constant c. After some simplification, we can show that

$$g(m) = O\left(\{2L \cdot (m+p+n)\}^{c \cdot (p+m) \cdot m^{c \cdot m}}\right)$$

where c is some other constant.

Proof of Lemma 3.5. Now consider the VASS G as given in Section 3. From the definition of G, it is easy to see that it satisfies the following properties: If c, c' are proper configurations appearing in a path of G then weight(c) = weight(c'). If c, c' are intermediate configurations appearing in a path of G and state(c) = state(c') then weight(c) = weight(c'). Now, let $\pi = c_0, c_1, \ldots, c_i, \ldots, c_{i+k}$ be a u-path of G such that c_0 is the initial configuration, $c_i \leq c_{i+k}$ and a final configuration appears between c_i and c_{i+k}. Since $c_i \leq c_{i+k}$, it is the case that state(c_i) = state(c_{i+k}). Since a final configuration of G is a proper configuration, it follows from (a) of Lemma 3.1 that c_{i+k} is a proper configuration or is an intermediate configuration. From our previous observations, it can be seen that $c_i = c_{i+k}$. From this we see that if π is an m-bounded self-covering u-path of G that starts with an initial configuration, then $\pi = \alpha\beta$ for some finite paths α and β that satisfy conditions (a), (b), and (c) of Lemma 3.4. From this, we see that for a finite path π of G, there exist α, β that satisfy conditions (a), (b), and (c) of Lemma 3.4 and such that $\pi = \alpha\beta$ iff π is an m-bounded self-covering u-path with a final state that starts from the initial configuration. Now for the VASS G, L is at most 2. Substituting this value for L, the following is easily seen. There exists a finite path $\alpha\beta$ where α, β satisfy the conditions (a), (b), and (c) of Lemma 3.4 iff there exists such a path of length

$$g(m) = O\left(\{2 \cdot (m+p+n)\}^{c \cdot (p+m) \cdot m^{c \cdot m}}\right),$$

where c is some constant.

The number n which is the number of transitions in G is $O(p^2 \cdot m^4)$. This can be seen as follows: For any transition t in G, there are at most two components in $vec(t)$; that are nonzero and the absolute value of each of them is bounded by 2, and the states source(t), target(t) can be any of the p states of G. From this it follows that n is $O(p^2 \cdot m^4)$ and hence is $O((p \cdot m)^4)$. From this, it is easy to see that $(m+n+p) = O((pm)^4)$. Using this, we get

$$g(m) = O\left((2pm)^{c \cdot (p+m) \cdot m^{c \cdot m}}\right),$$

where c is some other constant. Writing the above expression in a different

way, we get

$$g(m) = O(2^{c\cdot(p+m)\cdot\log(p+m)\cdot 2^{c\ m\ \log(m)}}).$$

By simplifying and using a different constant c, it can be shown that

$$g(m) = O(2^{c\cdot p\cdot\log(p)\cdot 2^{c\ m\ \log(m)}}).$$

Lemma 3.5 follows from this. □

Appendix C

For a VASS G and configurations c, c' of G, we write $G : c \Rightarrow c'$ to indicate that the configuration c' is reachable from the configuration c in G. If $\vec{a}$ is an m-vector of natural numbers, then let us define an *atomic condition* on $\vec{a}$ to be a predicate of one of the forms $\vec{a}[i] = 0$, $\vec{a}[i] = 1$, $\vec{a}[i] > 0$, or $\vec{a}[i] = \vec{a}[j]$. We define a *positive condition* on $\vec{a}$ to be a formula formed from atomic conditions and the propositional connectives $\wedge, \vee$. We say that a configuration $(s, \vec{a})$ satisfies a positive condition iff $\vec{a}$ satisfies the condition.

LEMMA C1. *Given a VASS G, a configuration c, and a positive condition f, it is decidable whether there exists a configuration c' such that $G : c \Rightarrow c'$, and c' satisfies f.*

PROOF. The problem can be reduced to the reachability problem for a VASS. That is, for each VASS G and positive condition F, we show how to construct a VASS G' with a fixed configuration c_{final} such that $G' : c \Rightarrow c_{\text{final}}$ iff for some configuration c' satisfying f, $G : c \Rightarrow c'$.

We begin by defining for each atomic condition ac, a VASS G_{ac} such that $G_{ac} : (\text{init}_{ac}, \vec{a}) \Rightarrow (\text{final}_{ac}, \vec{0})$, iff $\vec{a}$ satisfies the condition ac. The states init_{ac} (resp., final_{ac}) are the initial (resp., final) states of G_{ac}.

In order to describe the VASSes, we use a simple programming notation, which is most easily explained by example. We write

$$s_1 : \vec{0}[i := -1] \rightarrow s_2$$

to indicate that in state s_1, there is one transition, which has a vector that subtracts 1 from $\vec{a}[i]$, and then enters state s_2. We write

$$s_1 : \vec{0}[i := -1, j := -1] \rightarrow s_2$$

to indicate that the transition from s_1 to s_2 subtracts 1 from $\vec{a}[i]$ and $\vec{a}[j]$ $(i \neq j)$. We use the operator OR between descriptions of transitions to indicate that a state has more than one transition, and we write $\vec{0} \rightarrow s_1$ to indicate a transition with the vector $\vec{0}$. For example,

$$s_1 : \vec{0}[i := -1] \rightarrow s_2 \text{ OR } s_1 : \vec{0} \rightarrow s_3$$

indicates that there are two transitions from s_1, the second of which has the vector $\vec{0}$. Finally, we describe sets of transitions from a state by introducing a quantified variable, for instance

$$s_1 : \underset{k\,:\,k\neq i}{\text{OR}}\ \vec{0}[k := -1] \rightarrow s_2$$

means that for each value of k from 1 to the dimension of G, except for i, there is a transition that subtracts 1 from $\vec{a}[k]$ and enters s_2.

We can now describe the VASSes G_{ac}. For instance, to check the condition $\vec{a}[i] = 0$, the VASS $G_{\vec{a}[i]=0}$ repeatedly subtracts 1 from all vector components other than $\vec{a}[i]$. The condition $\vec{a}[i] = 0$ is true if the VASS can reach the vector $\vec{0}$. The transitions for G_{ac}, where ac is the condition $\vec{a}[i] = 0$ are

$$\text{init}_{ac} : \vec{0} \rightarrow \text{final}_{ac}$$

$$\text{final}_{ac} : \underset{k}{\text{OR}}\ \underset{k \neq i}{} \vec{0}[k := -1] \rightarrow \text{final}_{ac}.$$

For the condition $ac \equiv \vec{a}[i] = 1$, the transitions of G_{ac} are

$$\text{init}_{ac} : \vec{0}[i := -1] \rightarrow \text{final}_{ac}$$

$$\text{final}_{ac} : \underset{k \cdot k \neq i}{\text{OR}}\ \vec{0}[k := -1] \rightarrow \text{final}_{ac}.$$

For the condition $ac \equiv \vec{a}[i] > 0$, the transitions of G_{ac} are

$$\text{init}_{ac} : \vec{0}[i := -1] \rightarrow \text{final}_{ac}$$

$$\text{final}_{ac} : \underset{k}{\text{OR}}\ \vec{0}[k := -1] \rightarrow \text{final}_{ac}.$$

For the condition $ac \equiv \vec{a}[i] = \vec{a}[j]$, $i \neq j$, the transitions of G_{ac} are

$$\text{init}_{ac} : \vec{0}[i := -1, j := -1] \rightarrow \text{init}_{ac} \quad \text{OR} \quad \text{init}_{ac} : \vec{0} \rightarrow \text{final}_{ac}$$

$$\text{final}_{ac} : \underset{k : k \neq i, j}{\text{OR}}\ \vec{0}[k := -1] \rightarrow \text{final}_{ac}.$$

For the next step of the construction, we introduce a composition operation for forming VASSes. If G_1 and G_2 are any two VASSes, then let $G' = \langle G_1; G_2 \rangle$ denote the "sequential composition" of G_1 and G_2. This is formed by first renaming the states of G_2 to new names, distinct from G_1. Let $\tilde{G}_2$ be a VASS similar to G_2, except for this renaming. Then the set of states of G' is the union of states of $G_1, \tilde{G}_2$. The set of transitions of G' is the union of the transitions of $G_1, \tilde{G}_2$, together with new transitions from each final state of G_1 to the initial state of $\tilde{G}_2$, labeled with the vector $\vec{0}$. The initial state of G' is the initial state of G_1; the final states of G' are the final states of $\tilde{G}_2$.

Now, if f is an atomic condition, then the VASS $G' = \langle G; G_f \rangle$ with final state $\text{final}_{G'}$ has the property that $G' : c \Rightarrow (\text{final}_{G'}, \vec{0})$ iff for some c' satisfying f, $G : c \Rightarrow c'$.

If f is a positive condition but not atomic, then we first put it into disjunctive normal form $f_1 \vee \cdots \vee f_k$, where each f_i is a conjunction of atomic conditions. Then, we check whether there exists an f_i such that there is a path from configuration c to some configuration c' satisfying f_i. This determines whether there is path in G from c to some configuration satisfying f.

It remains to show how we check whether there is a path to a configuration that satisfies a conjunction of atomic conditions. The basic idea is that if there are n atomic conditions in the conjunction, then we modify G so that it makes n copies of each vector component in its configurations. Then for each atomic

condition, we use a VASS that sets one copy of the vector components to $\vec{0}$ if the condition is satisfied. We now make this precise.

If $\vec{a}$ is an m-vector of natural numbers, then let us define $n \cdot \vec{a}$ to be an $n \cdot m$ vector such that the first n elements are equal to $\vec{a}[1]$, the second n elements are equal to $\vec{a}[2]$, etc. Similarly, if $c = (s, \vec{a})$ is a configuration, then let $n \cdot c$ denote the configuration $(s, n \cdot \vec{a})$. It is clear that for any $n \geq 1$ and VASS G, we can form a VASS $n \cdot G$ such that $G : c \Rightarrow c'$ iff $n \cdot G : n \cdot c \Rightarrow n \cdot c'$.

Now, given a positive condition f which is a conjunction of atomic conditions, we show how to construct a VASS G' with the following properties:

(1) G' has the form $\langle n \cdot G; G_f \rangle$, where G_f is a VASS depending on f.
(2) There is a configuration c' such that $G : c \Rightarrow c'$ and c' satisfies f iff $G' : n \cdot c \Rightarrow (\text{final}_{G'}, \vec{0})$.

We construct G' by induction on the structure of f. For the base case, when f is an atomic condition, we have already given a construction of G' satisfying properties (1) and (2).

For the induction step, assume f has the form $f_1 \wedge f_2$. By the inductive hypothesis, there are VASSes of the form $\langle n_1 \cdot G; G_{f_1} \rangle$ and $\langle n_2 \cdot G; G_{f_2} \rangle$, which satisfy property (2) for f_1, f_2, respectively. Then, we form the VASS $G' = \langle (n_1 + n_2) \cdot G_1; \langle G^1_{f_1}; G^2_{f_2} \rangle \rangle$. Here, $G^1_{f_1}$ is a VASS with the same states as G_{f_1} and which operates on vectors that have $n_1 + n_2$ copies of each vector component of G. Note that G_{f_1} operates on n_1 copies of the vector components of G. The VASS $G^1_{f_1}$ updates the first n_1 copies of each component of the vectors of G in the same way that G_{f_1} does, and leaves the other components unchanged. Similarly, $G^2_{f_2}$ updates the last n_2 copies of each component of the vectors of G in the same way that G_{f_2} does, and leaves the others unchanged. Thus, G' has the property that $G : c \Rightarrow c'$ for some c' satisfying f iff $G' : (n_1 + n_2) \cdot c \Rightarrow (\text{final}_{G'}, \vec{0})$, as required. □

REFERENCES

1. Apt, K. R., and Kozen, D. Limits to automatic program verification. *Inf. Proc. Lett. 22*, 6 (1986), 307–309.
2. Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite state concurrent programs from temporal specifications. *ACM Trans. Prog. Lang. Syst. 8*, 2 (Apr. 1986), 244–263.
3. Clarke, E. M., and Grumburg, O. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, 1987, pp. 294–303.
4. Clarke, E. M., Grumberg, O., and Browne, M. Reasoning about networks with many identical finite state processes. In *Proceedings of 5th ACM Symposium on Principles of Distributed Computing* (Aug.). ACM, New York, 1986, pp. 240–248.
5. Emerson, E. A., and Lei, C. L. Modalities for model checking: Branching time strikes back. In *Proceedings of 12th ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 14–16). ACM, New York, 1985, pp. 84–96.
6. Emerson, E. A., and Sistla, A. P. Deciding full branching time logic. *Inf. Cont. 61*, 3 (June 1984), 175–201.
7. Halpern, J., and Vardi, M. The complexity of reasoning about knowledge and time. In *Proceedings of the 18th ACM SIGACT Symposium on Theory of Computing* (Berkeley, Calif., May 28–30). ACM, New York, 1986, pp. 304–315.

8. HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass., 1979.
9. KANELLAKIS, P. C., AND SMOLKA, S. A. On the analysis of cooperation and antagonism in networks of communicating processes. In *Proceedings of 4th ACM Symposium on Principles of Distributed Computing* (Minaki, Ont., Canada, Aug. 5–7). ACM, New York, 1985, pp. 23–38.
10. KARMARKAR, N. A new polynomial time algorithm for linear programming. In *Proceedings of the 16th ACM Symposium on Theory of Computing* (Washington, D.C., Apr. 30–May 2). ACM, New York, 1984, pp. 302–311.
11. KARP, R., AND MILLER, R. Parallel program schemata. *J. Comput. Syst. Sci. 3*, 4 (May 1969), 167–195.
12. KOSARAJU, S. R. Decidability of reachability in vector addition systems. In *Proceedings of 14th ACM Symposium on Theory of Computing* (San Fransico, Calif., May 5–7). ACM, New York, 1982, pp. 267–281.
13. KURSHAN, R. P., AND MCMILLAN, K. A structural induction theorem for processes. In *Proceedings of 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alberta, Canada, Aug. 14–16). ACM, New York, 1989, pp. 239–248.
14. LADNER, R. E. The complexity of problems in systems of communicating sequential processes. In *Proceedings of 11th ACM Symposium on Theory of Computing* (Atlanta, Ga., Apr. 30–May 2). ACM, New York, 1979, pp. 214–222.
15. LEHMANN, N. D. Knowledge, common knowledge, and related puzzles. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 62–67.
16. LICHENSTEIN, O., AND PNUELI, A. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of 12th ACM Annual Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 14–16). ACM, New York, 1985, pp. 97–107.
17. LIPTON, R. The reachability problem requires exponential space. Re. Rep. 62, Dept. of Computer Science, Yale Univ., January, 1976.
18. MANNA, Z., AND PNUELI, A. Specification and verification of concurrent programs by ∀-automata. In *Proceedings of 14th Annual ACM Symposium on Principles of Programming Languages* (Munich, West Germany, Jan. 21–23). ACM, New York, 1987, pp. 1–12.
19. MAYR, E. An algorithm for the general petri net reachability problem. In *Proceedings of 13th Annual ACM Symposium on Theory of Computing*. (Milwaukee, Wis., May 11–13). ACM, New York, 1981, pp. 238–246.
20. MILNER, R. A calculus of communicating systems. In *Lecture Notes in Computer Science*, Vol. 92. Springer-Verlag, New York, 1980.
21. PAPADIMITRIOU, C. H., AND STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
22. PETERSON, J. *Petri Net Theory and the Modeling of Systems* Prentice-Hall, Englewood Cliffs, N.J., 1981.
23. PETRI, C. Fundamentals of a theory of asynchronous information flow. In *Information Processing 62, Proceedings of the 1962 IFIP Congress*. North-Holland, Amsterdam, The Netherlands, 1962, pp. 386–390.
24. RACKOFF, C. The covering and boundedness problem for VAS. *Theoret. Comput. Sci. 6* (1978) 223–231.
25. REIF, J., AND SISTLA, A. P. A multiprocess network logic with temporal and spatial modalities. *J. Comput. Syst. Sci. 30*, 1 (Feb 1985), 41–53.
26. ROSIER, L., AND YEN, H. A multi-parameter analysis of the boundedness problem for VAS. *J. Comput. Syst. Sci. 32* (1986), 105–135.
27. SAVITCH, W. J. Relationship between non-deterministic and deterministic tape complexities. *J. Comput. Syst. Sci. 4*, 2 (1970), 177–192.
28 SISTLA, A. P. Theoretical issues in the design and verification of distributed systems. Ph.D. dissertation. Harvard Univ., Cambridge, Mass. 1983.
29. SISTLA, A. P., AND CLARKE, E. M. The complexity of propositional linear temporal logics. *J. ACM 32* (1985), 733–749.
30. SISTLA, A. P., AND GERMAN, S. M. Reasoning with many processes. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science* (June). IEEE, New York, 1987, pp. 138–152.
31. SISTLA, A. P., AND GERMAN, S. M. Reasoning with many processes Tech. Note. GTE Laboratories, Inc., 1987, (available on request from the authors).

32. SISTLA, A. P., VARDI, M., AND WOLPER, P. Complementation problem for Buchi automaton and its applications to temporal logic. *Theoret. Comput. Sci. 49*, 2, 3 (1987), 217–237.
33. VARDI, M., AND WOLPER, P. An automata theoretic approach to automatic program verification. In *Proceedings of 1st IEEE Symposium on Logic in Computer Science* (June). IEEE, New York, 1986.
34. VARDI, M., WOLPER, P., AND SISTLA, A. P. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science* (Tucson, Az.). IEEE, New York, 1983, pp. 185–194.
35. WOLPER, P., AND LOVINFOSSE, V. Verifying properties of large sets of processes with network invariants (preliminary Draft). In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*. Grenoble, France, June, 1989.

RECEIVED APRIL 1988; REVISED APRIL 1990; ACCEPTED DECEMBER 1990