Global and local views of state fairness*

Rodney R. Howell

Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA

Louis E. Rosier

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

Hsu-Chun Yen

Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, ROC

Communicated by A. Salomaa
Received January 1988
Revised October 1988

Abstract

Howell, R.R., L.E. Rosier and H. Yen, Global and local views of state fairness, Theoretical Computer Science 80 (1991) 77-104.

In this paper, we compare global and local versions of state fairness for systems of concurrent programs and Petri nets. We then investigate complexity and decidability issues for the fair nontermination problem. It turns out that for systems of concurrent Boolean programs and 1-bounded Petri nets, the problem is PSPACE-complete with respect to global state fairness, but EXPTIME-complete with respect to local state fairness. For general systems of concurrent programs, both the globally and locally state fair nontermination problems are undecidable. (In fact, they are Π_1-complete and Σ_1^1-complete, respectively.[1]) On the other hand, the problem is decidable for general Petri nets with respect to global state fairness, and undecidable (Σ_1-complete) with respect to local state fairness.

1. Introduction

In recent years, nondeterministic models, such as Petri nets, concurrent programs, communicating sequential processes (CSP), and networks of communicating finite-state machines (CFSMs), have been successfully used to represent many real-world

* This research was supported in part by U.S. Office of Naval Research Grant No. N00014-86-K-0763 and National Science Foundation Grant No. CCR-8711579.

[1] Σ_1^1 (Σ_1) stands for the first level of the analytical (arithmetic) hierarchy. Π_1^1 (Π_1) denotes the set of languages whose complements are in Σ_1^1 (Σ_1).

systems that involve parallel computations (see, e.g., [2, 14, 27, 34]). A major benefit of such (nondeterministic) models is that they often have a simpler structure than their deterministic counterparts (cf. [22, 30, 31]). On the other hand, due to the nature of nondeterminism, it is, in general, hard to analyze these systems. One of the fundamental issues concerning nondeterministic systems is the study of problems concerning their infinite behavior (e.g., the deadlock, lockout [20], starvation, and termination problems), especially when a "fairness constraint" is taken into account. Throughout the last decade, several notions of fairness with respect to these models have been proposed in conjunction with problems related to nondeterministic computations. In fact, multitudes of papers have been written on the subject. For example, in [4], Carstensen and Valk investigated the well-known dining philosophers problem, taking fairness into consideration. In [3, 15, 16], questions of fairness concerning Petri nets were examined. A hierarchy of notions of fairness for Petri nets can be found in [1]. Problems concerning the fair termination of finite-state concurrent systems can be found in [13, 23, 28, 36, 39]. Each of the above investigated the problem of determining whether a given system of finite-state concurrent programs will terminate under the assumption of some fairness constraint. With respect to finite-state concurrent programs more general problems (i.e., model checking) have also been extensively studied (cf. [7, 8, 38]). Definitions of fairness concerning networks of CFSMs can be found in [10]. A hierarchy of fairness constraints with respect to CSP was proposed by Manna and Pnueli [25].

In the study of nondeterministic systems, a very useful practice is to represent each system configuration by a "state". A state can be regarded as the result of taking a "snapshot" of the system at a given instant. Such a representation allows us to describe a concurrent system by a directed graph in which each node represents a system state and each edge indicates a transition (operation) of some program. As far as a single-program nondeterministic system is concerned, this definition of "state" is unambiguous. However, for a system with concurrency (i.e., a system consisting of several programs running in parallel), the concept of state can have either a "global" or "local" interpretation. A global state is a description of the entire system at some point in its execution. In contrast, a local state for a particular program describes only the information concerning the resources to which that program has access. That is, global states correspond to the view of a global observer while local states correspond to the view of a local observer.

So far, most of the research in concurrent systems deals with the concept of global states. One of the reasons is that global state spaces appear to allow us to have a clean view of the actions within a system. Furthermore, verification and specification methods for concurrent systems under a global state space representation have been well investigated in recent years. However, there are some cases (for example, systems with an arbitrary number of processes [37]) where a global observer cannot exist. For such cases, the notion of a global state is, in some sense, meaningless. A related problem is that online verification methods utilizing global states have implementation difficulties for certain distributed systems from the practical point of view. (For example, in a message-passing system a machine cannot "freeze" the computation of the system and then acquire the necessary information.) Consequently, it is worth taking a closer look at the issue of "global" vs. "local" from both theoretical and practical points of view.

The main contribution of this paper is to examine the above issue from the theoretical point of view. To achieve this goal, we investigate the global and local versions of *state fairness* with respect to the nontermination problem[2] for concurrent programs and Petri nets. The notion of state fairness (or an enhanced version called *extreme fairness*) due to Pnueli [28] is of interest because, as was shown in [28], it can be used to capture the essence of "probabilistic" computations. Informally speaking, a computation is said to be *state fair* iff whenever a transition from a state occurs infinitely often, *all* enabled transitions from that state must be executed infinitely often. The motivation is that if a state occurs infinitely often and, at each juncture, one tosses an unbiased coin to determine the next transition, then the probability of almost always neglecting some outgoing transition (enabled at that particular state) is zero. A similar concept has also been considered by Queille and Sifakis [29] (under the name of *fair choice from states*) to study transition systems and their fairness-related properties.

The main concern here is to study how the global and local views of state fairness will affect the complexity of the fair nontermination problem. Our study concentrates on four models of parallel computation—systems of concurrent Boolean programs, 1-bounded Petri nets, general systems of concurrent programs, and general Petri nets. A summary of our results is presented in Table 1. Some related issues of local vs. common (global) knowledge in distributing computing can be found in [18, 9, 12, 21]. See also [33] for related problems concerning games.

Table 1
Complexity of the nontermination problem (with respect to polynomial-time reductions).

	Global state fairness	Local state fairness
Systems of concurrent Boolean programs	PSPACE-complete	EXPTIME-complete
1-bounded Petri nets	PSPACE-complete	EXPTIME-complete
Systems of concurrent programs	Π_1-complete	Σ_1^1-complete
Petri nets	decidable	Σ_1-complete

The remainder of the paper is organized as follows. In Section 2, we provide an informal discussion about global and local versions of state fairness. In Section 3, we introduce the formal computational models utilized in this paper. Our complexity results with respect to systems of concurrent Boolean programs and 1-bounded Petri nets are presented in Section 4. Our results with respect to general systems are presented in Section 5.

[2] The problem of whether there exists an infinite computation that is state fair.

2. A discussion of the issue

In what follows, we explain the issue of global vs. local state fairness in an informal manner. A more formal description will be provided in Section 3. To understand the difference between global and local versions of state fairness, consider the example in Table 2 (see also Fig. 1 for the corresponding transition diagram). Here two programs A and B are running concurrently using x and y as shared variables (z is B's private variable). l_1 and l_2 (respectively m_1 and m_2) are program locations of program A (respectively program B). (Program locations are, conceptually, similar to statement labels in conventional programming languages.) Also

Table 2
A concurrent system.
$x = y = z = 0$

	Program A			Program B	
l_1:	**if** true **then** $y := 1$ **goto** l_1	$\dots a_1$	m_1:	**if** true **then** $y := 0$; $z := 0$ **goto** m_1	$\dots b_1$
	$\square$			$\square$	
	if $x = 1$ **then goto** l_1	$\dots a_2$		**if** $y = 1 \wedge z = 0$ **then** $y := 0$; $z := 1$ **goto** m_1	$\dots b_2$
				$\square$	
				if $y = 1 \wedge z = 1$ **then** $x := 1$ **goto** m_2	$\dots b_3$
l_2:	(Termination)		m_2:	(Termination)	

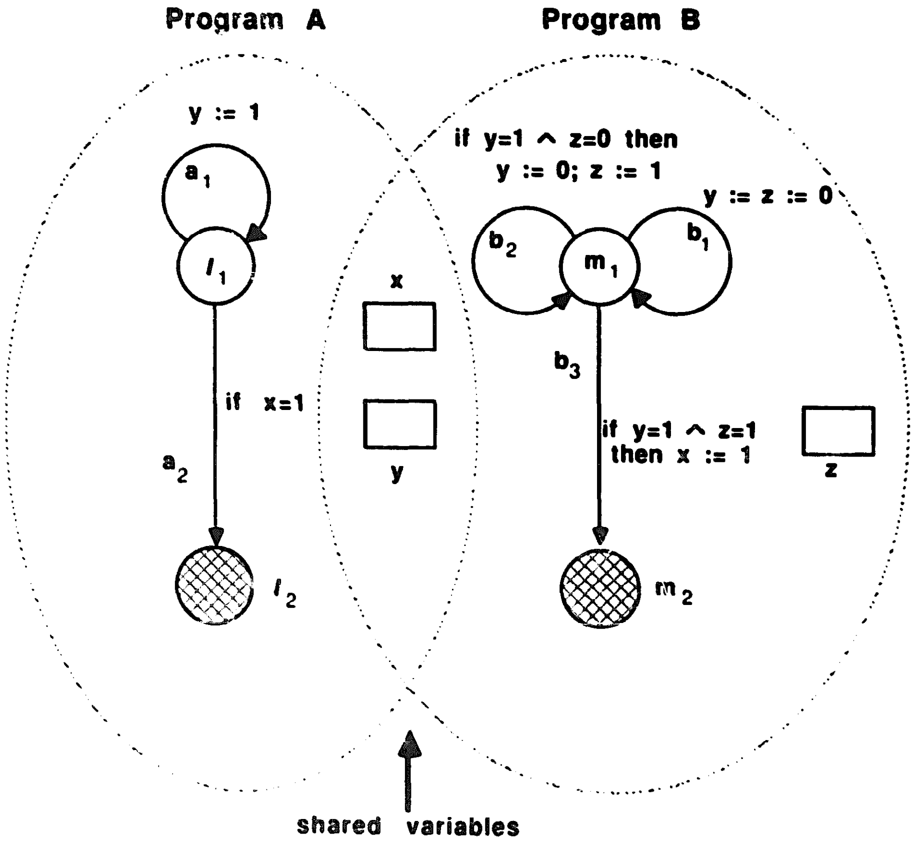


Fig. 1. A transition diagram for the concurrent system shown in Table 2.

notice that transitions a_1 and a_2 (respectively b_1, b_2 and b_3) of program A (respectively program B), separated by boxes in Table 2, are executed nondeterministically. We define a global state to be a tuple $[(l; x, y), (m; x, y, z)]$, where l and m are program locations of A and B, respectively, and x, y and z are variables; while local states of A and B are represented by $(l; x, y)$ and $(m; x, y, z)$, respectively. (More rigorous definitions of global and local states will be given in Section 3.)

Consider the infinite execution sequence σ: $[a_1, a_1, b_1, a_1, b_2, b_1, b_1]^\omega$ (a_i and b_i refer to transitions shown in Fig. 1 and the superscript ω indicates that the sequence will be repeated infinitely many times). Suppose that the job of an *observer* is to determine whether the above computation is state fair. The traditional view is that the observer is provided with a large observational window that views the entire system. However, such a view may not be feasible. Instead, one can have many observers, each having a small observational window which is capable of viewing the configuration of a single program. For example, suppose the current configuration (global state) of the system shown in Table 2 is $[(l; x, y), (m; x, y, z)]$. Two kinds of pictures, i.e.,

$$\boxed{(l; x, y)(m; x, y, z)} \quad \text{and} \quad \boxed{(l; x, y)} \quad \left(\text{or} \quad \boxed{(m; x, y, z)}\right),$$

can be viewed depending on the size of the observational window. In what follows, we shall see that the issue of whether the observer judges a computation to be state fair depends on the size of the observational window. Now, viewing σ through the large window, the observer can see the following (global) state sequence:

$$\begin{aligned}
&[[(l_1; 0,0), (m_1; 0,0,0)] \xrightarrow{a_1} [(l_1; 0,1), (m_1; 0,1,0)] \xrightarrow{a_1} [(l_1; 0,1), (m_1; 0,1,0)] \\
&\qquad \xrightarrow{b_1} [(l_1; 0,0), (m_1; 0,0,0)] \xrightarrow{a_1} [(l_1; 0,1), (m_1; 0,1,0)] \\
&\qquad \xrightarrow{b_2} [(l_1; 0,0), (m_1; 0,0,1)] \xrightarrow{b_1} [(l_1; 0,0), (m_1; 0,0,0)] \\
&\qquad \xrightarrow{b_1} [(l_1; 0,0), (m_1; 0,0,0)]]^\omega.
\end{aligned}$$

In this sequence, a_1 is never executed from the global state $[(l_1; 0,0), (m_1; 0,0,1)]$, although a_1 is enabled infinitely often in that state. Consequently, the computation is not state fair in the global sense. On the other hand, using smaller windows, observers will view the following two state sequences (depending upon whether the observer's view is focused on A or B) corresponding to σ:

$$\begin{aligned}
A\text{:}\quad &[(l_1; 0,0) \xrightarrow{a_1} (l_1; 0,1) \xrightarrow{a_1} (l_1; 0,1) \xrightarrow{b_1} (l_1; 0,0) \xrightarrow{a_1} (l_1; 0,1) \\
&\qquad \xrightarrow{b_2} (l_1; 0,0) \xrightarrow{b_1} (l_1; 0,0) \xrightarrow{b_1} (l_1; 0,0)]^\omega
\end{aligned}$$

and

$$\begin{aligned}
B\text{:}\quad &[(m_1; 0,0,0) \xrightarrow{a_1} (m_1; 0,1,0) \xrightarrow{a_1} (m_1; 0,1,0) \xrightarrow{b_1} (m_1; 0,0,0) \\
&\qquad \xrightarrow{a_1} (m_1; 0,1,0) \xrightarrow{b_2} (m_1; 0,0,1) \xrightarrow{b_1} (m_1; 0,0,0) \xrightarrow{b_1} (m_1; 0,0,0)]^\omega.
\end{aligned}$$

Transition a_1 occurs infinitely often in A's two local states $(l_1; 0, 0)$ and $(l_1; 0, 1)$. Also, transition b_1 (respectively b_2) occurs infinitely often in B's local states $(m_1; 0, 0, 0)$, $(m_1; 0, 0, 1)$ and $(m_1; 0, 1, 0)$ (respectively $(m_1; 0, 1, 0)$). As a result, the computation will be judged state fair by the observers using small windows; i.e., σ is state fair in the local sense. It is reasonably easy to see that there is no globally state fair infinite computation for this system. Hence, the example reveals that the answer to the state fair termination problem for concurrent systems might depend on the underlying notion of fairness (i.e., whether global or local fairness is assumed).

Consider also that in many real-world concurrent systems, the ratio of execution speeds between processes can be arbitrarily large (but finite). This too, in some cases, will affect the termination of a system. To see this, consider again the example in Table 2. Suppose the speed ratio of A to B is exactly reflected by the computation σ. In this case, transition a_1 was never ready when the global state $[(l_1; 0, 0), (m_1; 0, 0, 1)]$ was reached (i.e., when B's private variable z had a value of 1). (This can happen if, at that moment, A was still executing its internal operations.) Hence, a_1 will never be executed in that state. Therefore, such "collaboration" can prevent terminating computations. As a result, state fair computations for such a system might be those defined by local fairness.

As far as we know, no efforts have been made to clarify the fundamental issue of global vs. local state fairness. In fact, the underlying architecture of concurrent systems appearing in the literature is usually based on the concept of a global state space [13, 23, 29]. Although for some applications this approach is applicable, for others it may not be suitable. In addition to this, since each globally state fair nonterminating computation is also locally state fair, proving that a system will terminate with respect to local state fairness will guarantee the termination of the system in terms of global state fairness.

Based on the above observations, local vs. global state fairness is an issue that deserves further study from both theoretical and practical points of view. In Section 3, we define formally global and local versions of state fairness for two concurrent models, namely, systems of concurrent programs and Petri nets. In Section 4, we study the complexity of the nontermination problems for restricted versions of the above two models, i.e., systems of concurrent Boolean programs (i.e., programs with Boolean variables) and 1-bounded Petri nets (i.e., Petri nets with 1-bounded places). It turns out that for these restricted models, the globally state fair nontermination problem is complete[3] for PSPACE[4] (polynomial space), whereas the locally state fair nontermination problem is complete for EXPTIME[5] (exponential time).

[3] Completeness results mentioned in this paper, unless otherwise stated, are with respect to polynomial-time many-one reductions.

[4] PSPACE $= \bigcup_{c>0}$ DSPACE(n^c), where DSPACE$(S(n))$ denotes the class of languages accepted by deterministic Turing machines in $S(n)$ space.

[5] EXPTIME $= \bigcup_{c>0}$ DTIME(2^{n^c}), where DTIME$(T(n))$ denotes the class of languages accepted by deterministic Turing machines in $T(n)$ time.

In Section 5, we investigate the decidability of these problems with respect to general systems of concurrent programs and general Petri nets. We are able to show that for concurrent programs, the globally and locally state fair nontermination problems are undecidable (Π_1-complete and Σ_1^1-complete, respectively). On the other hand, for Petri nets the globally state fair nontermination problem is decidable, while the locally state fair nontermination problem is undecidable (Σ_1-complete). These results seem to indicate that despite the above merits for some applications, problems related to local state fairness are, in general, harder to analyze than those related to global state fairness. We also hope that the results of this study will allow us to have a better insight into the nature of parallel computations.

3. The models

In this section, we define global and local versions of the state fair nontermination problem for two concurrent models, namely, systems of concurrent programs and Petri nets.

Let Z denote the set of integers. A system of concurrent programs S is a triple (P, V, ν_0), where $P=\{P_1,\ldots,P_k\}$ is a finite set of *programs* (defined below), $V=\{v_1,\ldots,v_{h_0}\}$ is a finite set of *variables*, and $\nu_0: V\to Z$ is the *initial value function.* Each program P_i, $1\le i\le k$, is a 5-tuple $(Q_i, V_i, \delta_i, X_i, s_i)$, where

(1) $Q_i=\{r_1^i,\ldots,r_{d_i}^i\}$ is a finite set of *program locations* (the reader can think of program locations as statement labels, as in FORTRAN or PASCAL);

(2) $V_i=\{v_1^i,\ldots,v_{h_i}^i\}\subseteq V$;

(3) $\delta_i=\{\delta_1^i,\ldots,\delta_{d_i}^i\}$, where each δ_j^i, $1\le j\le d_i$, is a set of *transitions* at location r_j^i. Each transition $t\in\delta_j^i$ is of the following form:

$$\textbf{if } p(v_1^i,\ldots,v_{h_i}^i) \textbf{ then } x_1 := y_1+c_1;\ldots;x_m := y_m+c_m \textbf{ goto } r_z^i,$$

where p is a logical expression over the variables $v_1^i,\ldots,v_{h_i}^i$, integer constants, the arithmetic operator $+$, the relational operators $=$, $<$, and $>$, and the logical operators $\vee$, $\wedge$, and $\neg$; $\{x_1,\ldots,x_m,y_1,\ldots,y_m\}\subseteq V_i$; $c_1,\ldots,c_m$ are (possibly negative) integer constants; and $r_z^i\in Q_i$;

(4) $X_i\subseteq Q_i$ is the set of *terminal* program locations; and

(5) $s_i\in Q_i$ is the *initial* program location.

We also require that for $1\le i,j\le k$, if $i\ne j$, then $Q_i\cap Q_j=\emptyset$. (Note that this implies that $\delta_i\cap\delta_j=\emptyset$.) Figure 2 provides a pictorial description of a concurrent system.

The *size* of a concurrent system S or a program P_i (denoted by $\|S\|$ and $\|P_i\|$, respectively) is defined to be the length of its description when a standard binary encoding technique is used. Given a system S of k programs, the set of *shared variables* between programs P_i and P_j, $i\ne j$, is the set $G_{i,j}=V_i\cap V_j$. For a given P_i, $V_i-\bigcup_{j\ne i}G_{i,j}$ is the set of P_i's *private variables*. A *global state* of S is a pair $[\alpha,\nu]$, where $\alpha: P\to\bigcup_{i=1}^k Q_i$ such that $\alpha(P_i)\in Q_i$ is the *current location function*, and $\nu: V\to Z$ is the *current value function.* The *initial state* of S is the pair $[\alpha_0,\nu_0]$ such

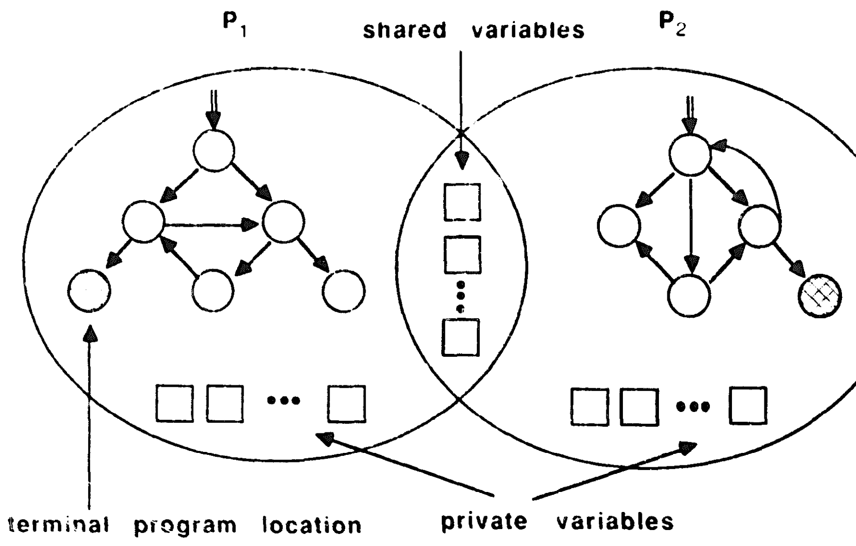


Fig. 2. A two-program concurrent system.

that $\alpha_0(P_i) = s_i$. For a program $P_i \in P$, the *local state* of P_i associated with the global state $[\alpha, \nu]$ is the pair $(\alpha(P_i), \nu(V_i))$, where $\nu(V_i)$ denotes the restriction of ν to V_i. A local state $(r'_i, \nu(V_i))$ is *terminal* iff $r'_i \in X_i$. A global state $[\alpha, \nu]$ is terminal iff there is an i such that $\alpha(P_i) \in X_i$. (Note that the sets of terminal locations are defined arbitrarily and have nothing to do with halting.)

Let $t \in \delta'_i$ be the transition

$$\textbf{if } p(v'_1, \ldots, v'_{k_i}) \textbf{ then } x_1 := y_1 + c_1; \ldots; x_m := y_m + c_m \textbf{ goto } r'_2.$$

t is said to be *enabled* at the global state $[\alpha, \nu]$ and the local state $(r'_1, \nu(V_i))$ iff $\alpha(P_i) = r'_1$ and $p(\nu(v'_1), \ldots, \nu(v'_{k_i})) = \text{true}$. We then write $[\alpha, \nu] \xrightarrow{t} [\alpha', \nu']$ and $(r'_1, \nu(V_i)) \xrightarrow{t} (r'_2, \nu'(V_i))$, where

- $\alpha'(P_i) = r'_2$;
- $\alpha'(P_{i'}) = \alpha(P_{i'})$ for $i' \neq i$;
- $\nu'(x_h) = \nu(y_h) + c_h$ for $1 \leq h \leq m$; and
- $\nu'(v_h) = \nu(v_h)$ for all other $v_h \in V$.

A *computation* is a sequence of global states

$$\sigma : [\alpha_0, \nu_0] \xrightarrow{t_1} [\alpha_1, \nu_1] \xrightarrow{t_2} \cdots \xrightarrow{t_i} [\alpha_i, \nu_i] \xrightarrow{t_{i+1}} \cdots.$$

A computation can be finite or infinite. A system $S = (P, V, \nu_0)$ is a system of *Boolean programs* if for all global states $[\alpha, \nu]$ in all computations, $\nu(v_i) \in \{0, 1\}$ for all $v_i \in V$.

A transition t is *globally state fair* (*locally state fair*) for a computation σ iff t is executed infinitely often in all the global (local) states at which it is enabled infinitely often. A computation is globally state fair (locally state fair) iff all of its transitions are globally state fair (locally state fair). Note that a globally state fair computation is also a locally state fair computation and that the definitions are identical for single-program systems. A *nonterminating* computation is an *infinite* computation in which no global state is a terminal state.

For convenience, we now introduce some alternative notation to that defined above. First, without increasing the expressive power, we can describe programs using high-level language constructs, such as **loop** ... **end-loop**, **if** ... **then** ... **else**. It is not hard to see that these constructs can easily be implemented using if-then-goto statements. We can then construct a system S from n programs so defined as long as all initial values of variables are clear from the context. We will then write $S = (P_1, \ldots, P_n)$. Also, if an ordering is assumed on the h_i variables in program P_i, we may represent a restriction of a current value function $\nu(V_i)$ as an h_i-dimensional vector $\bar{V}' = \langle \bar{v}_1, \ldots, \bar{v}_{h_i} \rangle$, where each $\bar{v}_j = \nu(v_j)$. A local state of P_i may then be represented as $(r_i', \bar{v}_1, \ldots, \bar{v}_{h_i})$ (or $(r_i'; \bar{V}')$). A global state of S may then be represented as $[q_1, \ldots, q_n]$, where each q_i is a local state of P_i. It should be noted, however, that given a local state q_i for each P_i, $[q_1, \ldots, q_n]$ does not necessarily represent a global state of S.

The *globally* (*locally*) *state fair nontermination problem* is to determine, given a system of concurrent programs, whether there exists a globally (locally) state fair nonterminating computation.

In the second part of this section, we define the globally and locally state fair nontermination problems with respect to Petri nets. We first give some preliminaries of Petri nets that are needed for the remainder of this paper. The reader should consult [27, 34] for more detailed definitions. A *Petri net* $\mathscr{C}$ is a 4-tuple (P, T, φ, μ_0), where P is a finite set of *places*, T is a finite set of *transitions*, φ is the *flow function* $\varphi : (P \times T) \cup (T \times P) \to N$, and μ_0 is the *initial marking* $\mu_0 : P \to N$, where N is the set of natural numbers. For each $t \in T$, we let $^{\bullet}t = \{p \mid \varphi(p, t) > 0\}$ and $t^{\bullet} = \{q \mid \varphi(t, q) > 0\}$ be the sets of input and output places of t, respectively. A *marking* is a mapping $\mu : P \to N$. We often establish an order on the places, $p_1, \ldots, p_k$, and designate a marking μ as a vector in N^k where the ith component represents $\mu(p_i)$. We say $\mu(p_i)$ is the number of *tokens* in place p_i at μ. A transition $t \in T$ is *enabled* at a marking μ iff for every $p \in P$, $\mu(p) \geq \varphi(p, t)$. If t is enabled at μ, we write $\mu \xrightarrow{t} \mu'$, where $\mu'(p) = \mu - \varphi(p, t) + \varphi(t, p)$ for all $p \in P$, to represent the action of *firing* t at μ. μ' is the resulting marking. A sequence of transitions $\sigma = t_1 t_2 \ldots t_n \ldots$ is a *firing sequence* from μ_0 iff $\mu_0 \xrightarrow{t_1} \mu_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \mu_n \xrightarrow{t_{n+1}} \cdots$. A firing sequence can be finite or infinite. (If $\sigma = t_1 t_2 \ldots t_n$ is finite we sometimes write $\mu_0 \xrightarrow{\sigma} \mu_n$.) A *computation* from μ_0 is a (finite or infinite) sequence of markings $\mu_0 \mu_1 \ldots \mu_n \ldots$ such that $\mu_0 \xrightarrow{t_1} \mu_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \mu_n \xrightarrow{t_{n+1}} \cdots$, for some firing sequence $t_1 t_2 \ldots t_n \ldots$. If there exists a finite firing sequence $\sigma = t_1 t_2 \ldots t_n$ such that $\mu_0 \xrightarrow{\sigma} \mu_n$, then μ_n is said to be *reachable* from μ_0 via σ. For a Petri net $\mathscr{C} = (P, T, \varphi, \mu_0)$, the *reachability set*, denoted by $R(\mathscr{C})$, is the set of markings $\{\mu \mid \mu_0 \xrightarrow{\sigma} \mu$, for some finite $\sigma\}$. Given a Petri net $\mathscr{C}$ and a marking μ, the *reachability problem* is to determine whether $\mu \in R(\mathscr{C})$. A Petri net $\mathscr{C}$ is said to be 1-*bounded* iff for every $\mu \in R(\mathscr{C})$, $\mu(p) \leq 1$, for all $p \in P$ (i.e., the number of tokens in any place will never exceed 1).

A *partition* of a Petri net (P, T, φ, μ_0) is a set of subsets of transitions $\mathscr{T} = \{T_1, \ldots, T_r\}$ which satisfies $T_i \cap T_j = \emptyset$, for all $i \neq j$, and $\bigcup_{i=1}^{r} T_i = T$. (Partition elements with respect to Petri nets correspond to programs in a system of concurrent programs.) Given a subset of transitions T', we let $\ddot{T}'$ be the set of places $\{p \mid \exists t \in T', \varphi(p, t) > 0$ or $\varphi(t, p) > 0\}$. Given a marking μ and a subset of places P' ($\subseteq P$), we use $\mu(P')$ to denote the restriction of μ to P'. ($\mu(P')$ is referred to as a submarking.) Given a computation $\sigma: \mu_0\mu_1 \ldots \mu_i \ldots$ and a subset of places P', we define the *projection* of σ on P', denoted by $\sigma(P')$, to be the sequence $\mu_0(P')\mu_1(P') \ldots \mu_i(P') \ldots$ Given a partition $\mathcal{T} = \{T_1, \ldots, T_r\}$, an infinite computation $\sigma: \mu_0\mu_1 \ldots \mu_i \ldots$ is said to be *state fair* with respect to $\mathcal{T}$ if it satisfies the following condition: $\forall i$, $1 \leq i \leq r$, $\forall$ marking μ, $\forall t \in T_i$ enabled at μ, if there exist infinitely many j's such that $\mu_j(\ddot{T}_i) = \mu(\ddot{T}_i)$, then t must be fired at infinitely many of these μ_j's.

The *locally state fair nontermination problem* for Petri nets is to determine, given a Petri net and a partition $\mathcal{T}$, whether there exists an infinite computation which is state fair with respect to $\mathcal{T}$. The *globally state fair nontermination problem* for Petri nets is to determine, given a Petri net $\mathcal{C} = (P, T, \varphi, \mu_0)$, whether there exists an infinite computation which is state fair with respect to $\{T\}$. (Note that $\mathcal{T}$ was ignored in the latter definition just as the set of programs was ignored in conjunction with global state fairness for systems of concurrent programs.)

4. Complexity results for systems of concurrent Boolean programs and 1-bounded Petri nets

In this section, we first establish the relationship between 1-bounded Petri nets and systems of concurrent Boolean programs by showing that one is computationally harder in a specific way. We then derive the complexity of the nontermination problems for 1-bounded Petri nets and systems of concurrent Boolean programs with respect to global and local state fairness.

Lemma 4.1. *Given a 1-bounded Petri net $\mathcal{C} = (P, T, \varphi, \mu_0)$ and a partition $\mathcal{T} = \{T_1, \ldots, T_r\}$, we can construct, in polynomial time, a system of r concurrent Boolean programs S such that $\mathcal{C}$ has a globally (locally) state fair nonterminating computation (with respect to $\mathcal{T}$) iff S has a globally (locally) state fair nonterminating computation.*

Proof. Let $\mathcal{C} = (P, T, \varphi, \mu_0)$ be a 1-bounded Petri net and $\mathcal{T} = \{T_1, \ldots, T_r\}$ a partition of T. One simply constructs a system of r concurrent Boolean programs $S = (P_1, \ldots, P_r)$ over $|P|$ variables as follows. Each variable will correspond to a unique place in P. Each P_i, $1 \leq i \leq r$, will consist of a single program location and have access to the variables that correspond to places in $\ddot{T}_i$. P_i, $1 \leq i \leq r$, will have $|T_i|$ transitions each designed to simulate a transition of T_i. The variables then are initialized so as to correspond to μ_0. The construction should now be obvious. Note that since $\mathcal{C}$ is 1-bounded, S is a system of Boolean programs. The lemma then follows immediately from the respective definitions of state fairness. □

We would like to show the converse of this lemma. That is, given a system of r concurrent Boolean programs S, that we can construct, in polynomial time, a 1-bounded Petri net $\mathscr{C} = (P, T, \varphi, \mu_0)$ and a partition $\mathscr{T} = \{T_1, \ldots, T_r\}$ such that S has a globally (locally) state fair nonterminating computation iff $\mathscr{C}$ has a globally (locally) state fair nonterminating computation (with respect to $\mathscr{T}$). Unfortunately, there seems to be a problem with such a construction. We would have to design a way for $\mathscr{C}$ to emulate the *if-then-goto* statements. Now suppose S (or a program in S) has access to n variables. Each Boolean predicate over these variables is a function from $\{0, 1\}^n$ to $\{0, 1\}$. Hence, we can write doubly exponential distinct such predicates. Because there are only singly exponential global states for S, however, only singly exponential outcomes may arise. Yet, it seems hard to devise a way for $\mathscr{C}$ to distinguish the action caused by a particular predicate—at least in polynomial time. As a result the converse of Lemma 4.1 appears questionable. With respect to the issues of complexity studied here, however, we will show that this does not matter.

The next lemma indicates that for 1-bounded Petri nets, the globally (locally) state fair nontermination problem is as hard as the reachability problem for 1-bounded Petri nets under polynomial time reductions. Since the reachability problem for 1-bounded Petri nets is PSPACE-complete [17], we will have shown that the globally (locally) state fair nontermination problems for systems of Boolean programs and 1-bounded Petri nets are PSPACE-hard.

Lemma 4.2. *For an arbitrary* 1*-bounded Petri net* $\mathscr{C} = (P, T, \varphi, \mu_0)$ *and marking* μ, *one can construct, in polynomial time, a* 1*-bounded Petri net* $\mathscr{C}' = (P', T', \varphi', \mu_0')$ *and a partition* $\mathscr{T}$ *in such a way that* $\mu \in R(\mathscr{C})$ *iff* $\mathscr{C}'$ *has a globally (locally) state fair nonterminating computation with respect to* $\mathscr{T}$.

Proof. The new Petri net $\mathscr{C}' = (P', T', \varphi', \mu_0')$ is constructed as follows (see Fig. 3).

(1) $P' = \{q', q'' \mid q \in P\} \cup \{c, q_{loop}\}$—all distinct.

(2) $\forall t \in T$, if $^{\bullet}t = \{p_1, \ldots, p_n\}$ and $t^{\bullet} = \{q_1, \ldots, q_m\}$, T' contains a transition t' where the sets of input and output places of t' are $\{p_1', \ldots, p_n', c, q_1'', \ldots, q_m''\}$ and $\{p_1'', \ldots, p_n'', c, q_1', \ldots, q_m'\}$, respectively. (Recall that $^{\bullet}t$ and $t^{\bullet}$ are the sets of input and output places of t, respectively.) Note that c serves as a control place in the sense that t' is firable only if c possesses a token.

(3) T' contains a transition t_c where $^{\bullet}t_c = \{c\}$ and $t_c^{\bullet} = \emptyset$ (i.e., t_c is used to remove the token in c).

(4) T' contains a transition t_{loop} where $^{\bullet}t_{loop} = \{q_{loop}\}$ and $t_{loop}^{\bullet} = \{q_{loop}\}$.

(5) Lastly T' contains a transition h where the set of input places of h is $\{r' \mid \mu(r) = 1\} \cup \{s'' \mid \mu(s) = 0\}$ and the set of output places of h is $\{q_{loop}\}$. (h is used to test whether μ is reached.)

(6) $\mu_0'(q') = \mu_0(q)$, $\mu_0'(q'') = 1 - \mu_0(q)$, $\forall q \in P$,
$\mu_0'(c) = 1$, and
$\mu_0'(q_{loop}) = 0$.

Finally, let $\mathscr{T} = \{T'\}$.

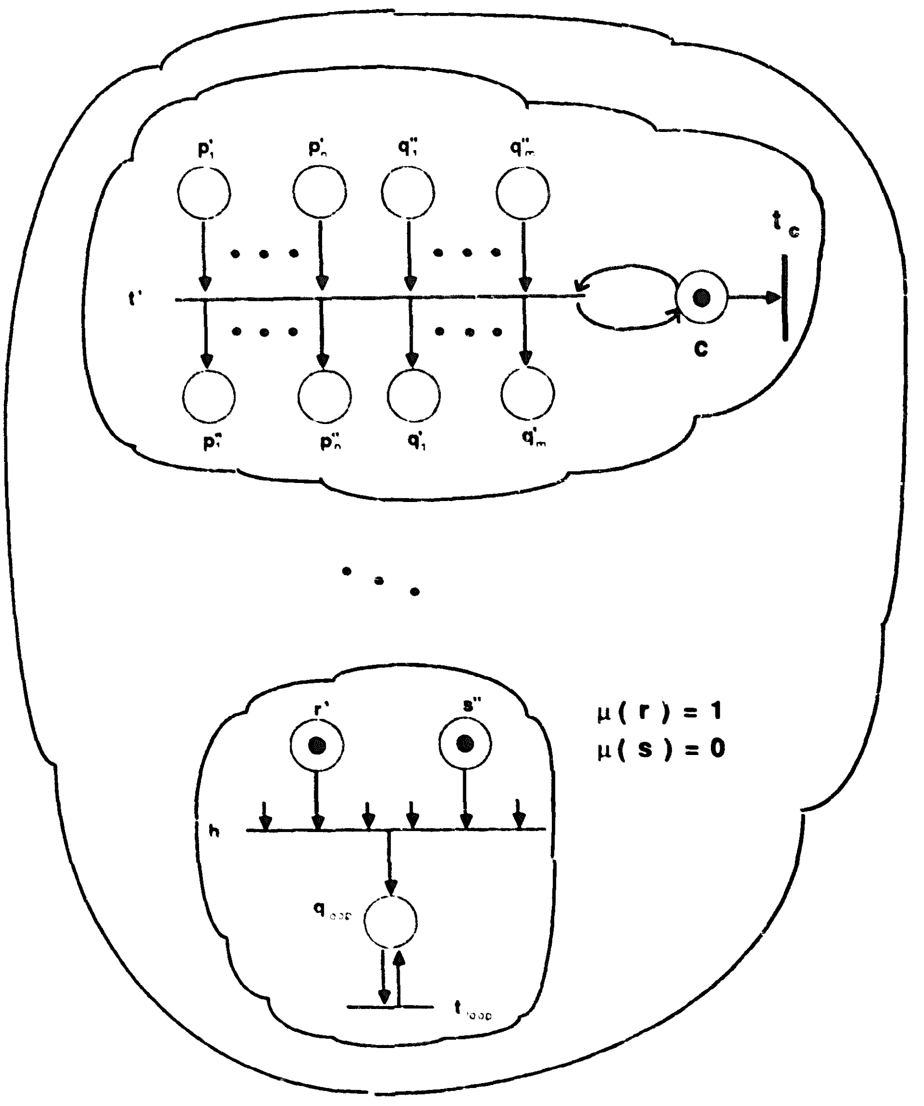


Fig. 3. Portions of a Petri net that simulates the reachability problem via the nontermination problem.

$\mathscr{C}'$ behaves as follows. As long as h has not fired $\mathscr{C}'$ mimics the moves of $\mathscr{C}$. During this time the current marking being simulated is represented by the number of tokens in the primed places (i.e., the current value of $q \in P$ is in $q' \in P'$). Also, during this time the total number of tokens in q'' and q' together is always exactly 1. Whenever t_c fires, c's token is removed and cannot subsequently be replaced. This action disables any further simulation of $\mathscr{C}$ by $\mathscr{C}'$. Thus, after t_c fires the only possible enabled transition is h. But h is only enabled if the number of tokens in the primed places corresponds to μ. Hence, if $\mu \in R(\mathscr{C})$ then there is a firing sequence for $\mathscr{C}'$ that can result in the firing of t_c, then h. Afterward, only t_{loop} can be enabled. This firing sequence can be extended by firing t_{loop} infinitely many times. It is now easy to see that this extended firing sequence defines a globally (locally) state fair nonterminating computation (with respect to $\mathscr{T}$). This takes care of the only if part.

Now suppose that $\mathcal{C}'$ has an infinite computation σ which is globally (locally) state fair with respect to $\mathcal{T}$. Clearly, t_i must fire in σ. Thus, if h does not fire, σ will be finite. But σ is infinite. Therefore, h must fire in σ—and this can only happen if $\mu \in R(\mathcal{C})$. This completes the proof. $\square$

In what follows, we show that the globally (locally) state fair nontermination problem is complete for PSPACE (EXPTIME) for systems of concurrent Boolean programs and 1-bounded Petri nets. This suggests that problems related to local fairness are, in some cases, harder to analyze than those related to global fairness. We first introduce some terminology.

Given a system of concurrent programs $S = (P_1, P_2, \ldots, P_k)$, the *global state graph*, denoted by G_S, is a directed labelled graph in which:

- each node in G_S is a k-tuple $[q_1, q_2, \ldots, q_k]$ which represents a global state of S, and
- there exists an edge labelled t from node $[q_1, \ldots, q_k]$ to node $[q'_1, \ldots, q'_k]$ in G_S iff $[q_1, \ldots, q_k] \xrightarrow{t} [q'_1, \ldots, q'_k]$.

Given two nodes s and s' in G_S, we use $s \rightsquigarrow s'$ to denote that there is a path from s to s' in G_S. Let s_0 be the initial state. A subgraph G' of G_S is *reachable* iff $s_0 \rightsquigarrow s'$ in G_S, for some node $s' \in G'$. A finite subgraph G' of G_S is called a *g-knot* iff

(1) G' is a strongly connected component (i.e., if $s, s' \in G'$, then $s \rightsquigarrow s'$ and $s' \rightsquigarrow s$ in G') having at least one edge,

(2) no node in G' is a terminal node (a node representing a terminal state), and

(3) $\forall$ transition t, if t is enabled in a node (state) s in G', then there is an edge (s, s') labelled t in G'. (An equivalent definition is that, $\forall s \in G'$, if $s \rightsquigarrow s'$, then $s' \in G'$.)

G' is an *l-knot* iff

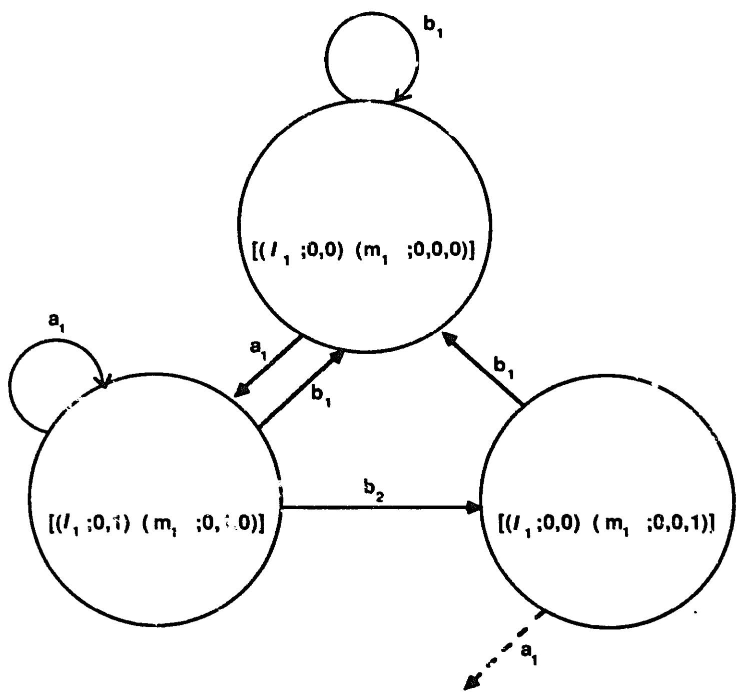
(1) G' is a strongly connected component having at least one edge,

(2) no node in G' is a terminal node, and

(3) $\forall$ transition t, if t is enabled in a node (state) $[q_1, \ldots, q_i, \ldots, q_k]$ in G', where P_i is the program containing t, then there exists in G' an edge (u, v) labelled t, where $u = [u_1, \ldots, u_i, \ldots, u_k] \xrightarrow{t} v = [v_1, \ldots, v_k]$ and $q_i = u_i$.

Note that in the definition of the l-knot, no edge labelled t in G' is required to originate at $[q_1, \ldots, q_i, \ldots, q_k]$, as was the case in the definition of the g-knot. Figure 4 is an l-knot with respect to the state graph corresponding to the system in Table 2. (The dashed edge does not belong to the graph.) Note that it is not a g-knot since transition a_1 (indicated by the dashed edge) is enabled in $[(l_1; 0, 0), (m_1; 0, 0, 1)]$, but not included in the subgraph as an outgoing edge of $[(l_1; 0, 0), (m_1; 0, 0, 1)]$; however, it is included as an outgoing edge of *some* node containing the local state $(l_1; 0, 0)$.

g- and l-knots for Petri nets can be defined similarly. It is worth mentioning that, conceptually, the notion of g-knots and that of so-called *traps* (defined in [11]) are similar. (A trap is a subset of places with the property that if, initially, it contains at least one token, then it cannot become empty (i.e., contain no tokens) by firing



b_1
$[(l_1\ ;0,0)\ (m_1\ ;0,0,0)]$
a_1
a_1
b_1
b_1
b_2
$[(l_1\ ;0,1)\ (m_1\ ;0,1,0)]$
$[(l_1\ ;0,0)\ (m_1\ ;0,0,1)]$
a_1

Fig. 4. An l-knot in the state graph of the system shown in Table 2.

transitions.) Both have the property that once a computation "falls" into a "blackhole" (a g-knot or a trap), the computation *must* remain in the blackhole forever. The fundamental difference between g-knots and traps for Petri nets is that g-knots are defined on the reachability graph of a Petri net; while traps are defined on the Petri net structure.

Before we derive the complexity result for the globally state fair nontermination problem, we first prove the following lemma, which provides a characterization of those global state graphs that admit globally state fair nonterminating computations.

Lemma 4.3. *Given a system of concurrent Boolean programs S (or a* 1*-bounded Petri net and a partition $\mathcal{T}$), there is a globally state fair nonterminating computation iff G_S has a reachable g-knot.*

Proof. Suppose G' is a reachable g-knot of G_S. Let s_0 be the initial node (state). According to the definition of a reachable g-knot, there exists a node s' in G' such that $s_0 \rightsquigarrow s'$ in G_S. Furthermore, there exists a path $s' \to s_1 \to \cdots \to s_d \to s'$ in G', for some d, which contains every node and utilizes every edge in G'. It is then easy to see that $s_0 \rightsquigarrow [s' \to s_1 \to \cdots \to s_d \to s']^\omega$ is a nonterminating globally state fair computation of S.

Now suppose l is a nonterminating globally state fair computation of S. Let G' be a subgraph of G_S consisting of those states and transitions that occur infinitely often in l. Clearly, G' is a strongly connected component. Since l is nonterminating, G' does not contain any terminal node. Furthermore, since l is globally state fair, any transition that is enabled in a state in G' must be executed infinitely often in l. Hence, the transition is in G'. Consequently, G' satisfies the three conditions of being a g-knot. This completes the proof $\square$

Now, given a system of k concurrent Boolean programs $S=(P_1,\ldots,P_k)$ of size n, the number of distinct global states can be as many as 2^{c*n}, for some fixed constant c. Hence, the size of the corresponding global state graph is, in general, $\Omega(2^{c*n})$—and always $O(2^{c*n})$. One might expect therefore that an algorithm to decide the globally state fair nontermination problem might require an exponential amount of space or time. In what follows, we show that despite the size of the global state graph, we need only space polynomial in n to solve the globally state fair nontermination problem for systems of concurrent Boolean programs. The next lemma provides a method to test for the existence of a g-knot without actually generating the global state graph or the g-knot.

Lemma 4.4. *Suppose G_S is finite. Then G_S has a reachable g-knot iff there is a reachable node s in G_S such that*

(1) *there is no terminal node reachable from s, and*

(2) *there is no deadend node (i.e., a node with no successors) reachable from s.*

Proof. According to the definition, a subgraph $G=(V,E)$ of G_S is a reachable g-knot iff it satisfies the following conditions:

(a) $\exists s\in G$ such that $s_0 \rightsquigarrow s$ in G_S,

(b) G is a strongly connected component with at least one edge,

(c) G contains no terminal node, and

(d) $\forall s\in G$, if $s\rightsquigarrow s'$ in G_S, for some s', then $s'\in G$.

Now, we first show the only if part, (i.e., if a reachable g-knot exists, then there is a reachable node s satisfying conditions (1) and (2)). Let $G=(V,E)$ be such a g-knot. Let s be any node in V. Suppose s can reach (in G_S) a deadend state or a terminal state, say s'. Let $s\to s_1\to\cdots\to s'$ be such a path. Due to condition (d), $s'\in V$. Since this conclusion contradicts either (b) or (c), the only if part holds.

On the other hand, suppose there is a node s satisfying both (1) and (2). Let Q be the set of all nodes reachable from s. Recall that $u\rightsquigarrow v$ ($u\to v$) denotes that v can be reached from u (in one step). We define an equivalence relation "$\sim$" such that $u\sim v$ iff $u\rightsquigarrow v$ and $v\rightsquigarrow u$. Using "$\sim$", one can decompose Q into equivalence classes. Let Q' be an equivalence class which has no successors with respect to "$\to$". (The finiteness of G_S, and hence Q, guarantees the existence of such a Q'.) Now, Q' cannot contain just a single state s' having no outgoing edges since such an s' would be a deadend node. Let $E=\{(u,v)\mid u\in Q'$ and (u,v) is an edge in $G_S\}$. We claim that the subgraph $G=(Q',E)$ is a reachable g-knot. Condition (a) follows immediately from the definition of Q. Since no terminal node is reachable from s, (c) is satisfied. Also, (b) and (d) are satisfied because Q' is an equivalence class having no successors with respect to "$\to$". This completes the proof of the if part. $\square$

Theorem 4.5. *The following two problems are* PSPACE-*complete*:
(1) *the globally state fair nontermination problem for systems of concurrent Boolean programs,*
(2) *the globally state fair nontermination problem for* 1*-bounded Petri nets.*

Proof. Since the reachability problem for 1-bounded Petri nets is PSPACE-complete [17], the lower bound for (1) and (2) follows directly from Lemmas 4.1 and 4.2. Hence, we need only consider the upper bound, and then only with respect to systems of concurrent Boolean programs (Lemma 4.1). But the upper bound follows directly from the characterization provided by Lemma 4.4. □

Recall that for systems consisting of a single program the notions of local and global state fairness are identical. In what follows, we show that the locally state fair nontermination problem is EXPTIME-complete for systems consisting of more than one concurrent Boolean program (or for 1-bounded Petri nets when the size of the partition is greater than 1). In what follows we show that an arbitrary polynomially space bounded *Alternating Turing Machine* (ATM) can be simulated, in some sense, by the locally state fair computations of a system of two concurrent Boolean programs. The lower bound is then obtained since polynomially space bounded ATMs have the same computational power as exponential time bounded deterministic Turing machines [6] (i.e., the class of machines defining EXPTIME).

An ATM M is a 5-tuple $(Q, \Sigma, \delta, q_0, g)$, where
- Q is a finite set of states,
- Σ is a finite tape alphabet (without loss of generality, we assume that the input and worktape alphabets are identical),
- $\delta \subseteq (Q\times\Sigma)\times(Q\times\Sigma\times\{-1, 0, +1\})$ is the next move relation,
- q_0 is the initial state,
- $g: Q \to \{$**existential, universal, accepting, rejecting**$\}$.

Basically the concept of alternation is a generalization of nondeterminism in a way that allows existential and universal quantifiers to alternate during the course of a computation. Four kinds of states exist in an ATM; namely existential, universal, accepting and rejecting states. ATM configurations, likewise, fall into one of the same four categories—depending solely on the current state. A universal configuration leads to acceptance iff all successor configurations lead to acceptance. An existential configuration leads to acceptance iff there exists a successor configuration that leads to acceptance. An ATM accepts its input iff the initial configuration leads to acceptance. Basically, the computation of an ATM is a tree. A path in this tree is called a *computation path*. During the course of a computation path, the segment between two consecutive alternations between types of configurations is called an *alternation block*. Detailed definitions can be found in [6]. The complexity classes of languages accepted by space (time) bounded ATMs were also defined in [6]. In particular, APSPACE is the set of languages accepted by polynomially space bounded ATMs. It was shown in [6] that APSPACE = EXPTIME. In what follows, this result will be used to prove the EXPTIME lower bound. Without loss of generality, we require that our polynomially space bounded ATMs:

(1) have initial configurations which are existential,
(2) be such that each computation path culminates in either an accepting or rejecting configuration, and
(3) be such that the number of successors of any configuration be 0 if the configuration is accepting or rejecting and 2 otherwise.

If M is a polynomially space bounded ATM that does not satisfy these properties an equivalent ATM M' that does can readily be constructed using standard techniques (see [6]).

To show the upper bound, we need the following easily shown lemma. Since the proof is very similar to that of Lemma 4.3, we leave it to the reader.

Lemma 4.6. *Given a system of concurrent Boolean programs S (or a 1-bounded Petri net and a partition $\mathcal{T}$), there is a locally state fair nonterminating computation iff G_S has a reachable l-knot.*

Theorem 4.7. *The following two problems are complete for* EXPTIME:

(1) *the locally state fair nontermination problem for systems of concurrent Boolean programs,*
(2) *the locally state fair nontermination problem for 1-bounded Petri nets.*

Proof. Because of Lemma 4.1, it will be sufficient to establish the lower (upper) bound with respect to (2) ((1)). We do illustrate the upper bound with respect to (1). However, we choose to illustrate the lower bound with respect to (1) also. We do this for what we feel is a very good reason. The lower bound proof is somewhat tedious and is much easier to understand in terms of Boolean programs. The same general idea works with respect to 1-bounded Petri nets but explanations thereof tend to become overly concerned with the ATM encodings. Getting a 1-bounded Petri net to simulate an ATM in the same fashion (as the Boolean programs do) is not difficult, but it does add significantly to the technical detail. The interested reader should consult [17], where simulations of LBAs (linear bounded automata) via 1-bounded Petri nets are discussed. The generalization to ATMs should be clear once the general strategy is understood.

Let M be an $h(n)$ space bounded ATM, where h is a polynomial function. Let x be an input for M. Let $|x| = n$. In what follows, we will show how to construct (in polynomial time) a system $S = (P_1, P_2)$ of two concurrent Boolean programs that will "simulate" the computation of M on x in such a way that S will have a locally state fair nonterminating computation iff M accepts x. Without loss of generality, we assume that M operates over a binary alphabet.

Basically, P_1 *repeatedly* simulates a computation path of M on x as long as each simulation path culminates in an accepting configuration. Each repetition of this simulation is referred to as a period. $h(n)$ local variables in P_1 will be used to simulate the contents of M's worktape. $\log h(n)$ (plus some constant number of) local variables of P_1 will be used to record M's current tapehead position (state, etc.). During different periods, P_1 may simulate different computation paths of M on x. If when simulating a computation path an accepting configuration is reached, P_1 reinitializes its variables in order to simulate another computation path. If, on the other hand, a rejecting configuration is reached, P_1 terminates.

Consider now a computation path (of M on x) that passes through a configuration q. Suppose that during an infinite computation σ of S, P_1 enters infinitely often a local state where the simulation being performed by P_1 is at q. If q is a universal configuration, P_1 should be enabled to advance the simulation to either of q's successor configurations. P_2 in conjunction with P_1 will be constructed so as to allow this. Thus in this case, providing σ is locally state fair, P_1 will infinitely often enter local states where the simulation being performed by P_1 is at each of q's successor configurations. If q is instead an existential configuration, P_1 should be enabled to advance the simulation to only one of q's successor configurations—the one that leads to acceptance. P_2 in conjunction with P_1 will be constructed so as to allow this. So, in this case, if σ is locally state fair P_1 need only enter infinitely often one of the two possible local states where the simulation being performed by P_1 is at a successor of q. Therefore overall, if σ is locally state fair, P_1 will infinitely often enter only and exactly those local states where the simulation being performed by P_1 is at a configuration of M on x that leads to acceptance. Since along σ, P_1 must infinitely often enter a local state where the simulation being performed is at the initial configuration of M on x, M must accept x. Likewise, if M does not accept x, σ cannot be locally state fair.

In what follows, we dub a local state of P_1 accepting, rejecting, universal, or existential depending on the category of the current configuration of M on x. Now the simulation of a computation path by P_1 (a period) proceeds in phases corresponding to the alternation blocks in the path. When P_1 is in a universal state, P_2 will be busy waiting. A shared variable D, set to 0, will insure this. At this time, P_1 will be enabled to simulate either of M's available moves. When P_1 enters a state that is existential (from any state), control is passed to P_2 by setting D to 1. P_1 is then busy waiting until P_2 sets D back to 0. P_2 then sets D to 0 and f nondeterministically to either 0 or 1, and resumes busy waiting. P_1 will now be enabled to simulate one of M's available moves if $f=0$ and the other if $f=1$. Thus, P_1 in an existential state will always be enabled to simulate exactly one but not both of M's available moves. P_2 (via its nondeterministic setting of f) simply controls which one becomes enabled. Finally, once the simulation of P_1 reaches an accepting configuration of M on x, P_1 sets D to 1, P_2 then sets f to 0 or to 1 (P_1 does not use this value of f; rather, this step is included only in order to ensure that the computation can be locally state fair with respect to P_2), then P_2 sets D to 0 and the entire procedure begins anew. As a result, $S=(P_1, P_2)$ behaves in the desired fashion. A detailed description of P_1 and P_2 now appears in Table 3. The detailed proof showing that S behaves as described is left to the reader.

Table 3
The concurrent system (P_1, P_2).

```
Shared variables: $D, f$.

 Program $P_1$:
 $L1$: Initialization; /Set the current configuration being simulated
                        to the initial configuration of $M$ on $x$./
     EU-flag := 0; /0 and 1 denote existential and universal phases, respectively./
     $D := 0; f := 0;$
     loop
     /Assume that the current configuration of $M$ on $x$ is $q$ and $q \to q'$
      and $q \to q''$ are its left and right transitions, respectively./
       if EU-flag = 0 then
         begin
           $D := 1$; /enable $P_2$/
 $L_\exists$:     if $D = 1$ then goto $L_\exists$ /busy-waiting/;
           if $D = 0 \wedge f = 0$ then "simulate $q \to q'$";
           if $D = 0 \wedge f = 1$ then "simulate $q \to q''$";
         end
       else
 $L_\forall$:     "pick $q \to q'$ or $q \to q''$ to simulate nondeterministically"
 $L_N$:   case current configuration /now either $q'$ or $q''$/
           accepting: goto $L2$;
           rejecting: TERMINATE;
           universal: EU-flag := 1;
           existential: EU-flag := 0;
         end case
     end loop
 $L2$: $D := 1$;
 $L3$: if $D = 0$ then goto $L1$ else goto $L3$;
 end

 Program $P_2$:
      loop
wait:   if $D = 0$ then goto wait;
        begin /nondeterministic choice/
          $f := 0$
        □ $f := 1$
        end
        $D := 0$;
      end loop
```

It should be noted that a locally state fair computation of S need not be globally state fair. The reason is that in order for existential moves to be simulated "correctly" infinitely often, P_2 must be able to set f "correctly" each time. This may not be possible if the computation is required to be globally state fair.

Now, we will show the upper bound with respect to (1) In a system S of n concurrent Boolean programs, the number of global states is $O(2^{c*n})$, for some fixed constant c. Hence, in exponential time we can construct the corresponding global state graph G_S. Recall that each node in G_S represents a global system state and that each edge in G_S represents a transition of one of the Boolean programs. Recall also that S will have a locally state fair nonterminating computation iff G_S has a reachable l-knot (Lemma 4.6). The following procedure can then be used to determine the existence of a reachable l-knot in G_S:

```
Algorithm: Partition $G_S$ into maximal strongly connected components (SCCs) $Q = \{G_1, \ldots, G_d\}$.
  while $Q \neq \emptyset$ do
    pick one element from $Q$, say $G$, and let $Q = Q - \{G\}$
    if $G$ is reachable from $s_0$
    then
      if $G$ is an $l$-knot,
      then output "$G_S$ has a reachable l-knot" and halt
      else
        (i) let $G'$ be the subgraph obtained from $G$ by removing all nodes and
            associated edges that violate the definition of an $l$-knot
        (ii) partition $G'$ into maximal SCCs $G'_1, \ldots, G'_f$
        (iii) let $Q = Q \cup \{G'_1, \ldots, G'_f\}$
  end-while
output "$G_S$ has no reachable l-knot".
```

The basic idea of the algorithm is the following. An l-knot is a special type of SCC. So to test whether an l-knot exists in G_S, we first decompose G_S into its set of maximal SCCs. Then, an arbitrary SCC G is chosen to test whether it is an l-knot. If it is, the procedure terminates; otherwise, the set Q is refined and the test is repeated. The fact that the algorithm works should almost be clear from the definitions. The only point that might need clarification is step (i). If G is not an l-knot, then there must exist either:

- a terminal node q_1 in G, or
- a node q_1 in G, a node q_2 outside of G, and a transition t (belonging to a program, say P_i) such that $q_1 \xrightarrow{t} q_2$. Furthermore, t does not label any edge in G emanating from a node whose local state (with respect to P_i) is the same as that of q_1.

(Note that by definition one of these items must exist.) Let G' be the subgraph of G resulting from the removal of q_1 and its incident edges. Clearly $G' \neq G$. Now if G contains an l-knot, the l-knot cannot contain q_1—hence it must be contained within an SCC of G'. The fact that the algorithm works should now be obvious.

Now we are ready to analyze the algorithm's complexity. First note that each of the following two steps can be carried out in time polynomial in the size of G_S:

- constructing the set of maximal SCCs, and
- determining whether a subgraph is an l-knot.

Therefore, the execution time required for the algorithm is polynomial in the size of G_S. Since the size of G_S is bounded by 2^{c*n}, this establishes the upper bound—and thus completes our proof. □

5. Decidability results for general systems

In this section, we investigate decidability issues of the globally (locally) state fair nontermination problems for general Petri nets and systems of concurrent programs. We will show that both the globally and locally state fair nontermination problems are undecidable for systems of concurrent programs. In particular, we show that the locally state fair nontermination problem is complete for Σ_1^1—the first level of the analytical hierarchy (see, e.g., [35])—whereas the globally state fair nontermination problem is complete for Π_1—the set of languages whose complements are accepted by Turing machines. With respect to Petri nets, the problem is decidable for global state fairness, but still undecidable for local state fairness. In particular, we show that the locally state fair nontermination problem for Petri nets is complete for Σ_1—the set of languages accepted by Turing machines. (The results here with respect to Petri nets should be compared and contrasted with those of [16] where each undecidable fair nontermination problem was Σ_1^1-complete rather than Σ_1-complete.) The disparity between concurrent systems and Petri nets is mainly because Petri nets operate in a more asynchronous fashion due to the lack of zero testing capabilities.

We first reproduce the following theorem, which we first showed in [16].

Theorem 5.1 (from [16]). *The globally state fair nontermination problem for Petri nets is decidable.*

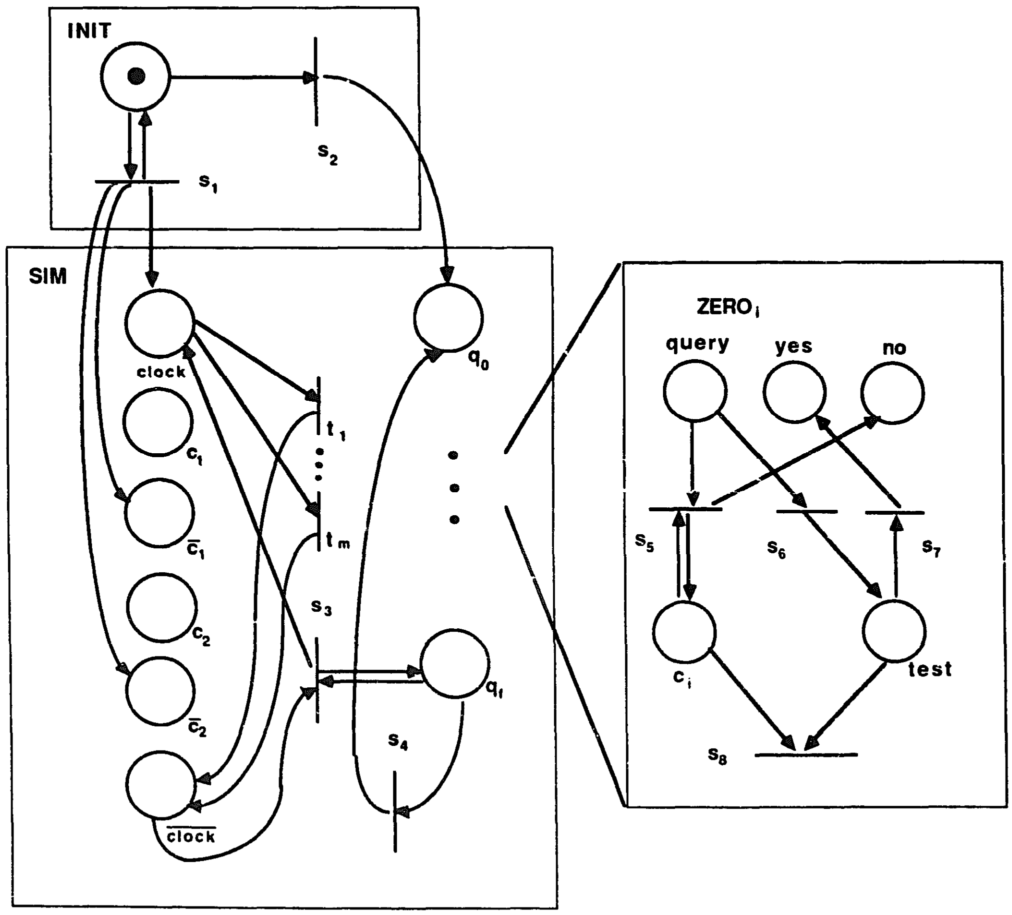
Proof. Let $\mathscr{C}$ be an arbitrary Petri net. We first determine whether $\mathscr{C}$ is bounded. (See [19, 32] for boundedness algorithms.) If $\mathscr{C}$ is unbounded, there is an infinite firing sequence σ which reaches each marking at most once. σ is clearly globally state fair. On the other hand, if σ is bounded, we can construct the reachability graph. Then there is an infinite state fair firing sequence iff the reachability graph contains a g-knot (i.e., a strongly connected component from which there is no exit). □

In what follows, we show the locally state fair nontermination problem with respect to Petri nets to be complete for Σ_1. The typical strategy for showing fair nontermination problems (with respect to Petri nets) to be undecidable is to show that the fairness constraint can be used to eventually force correct zero-testing in the simulation of a counter machine (see, e.g., [3]). This strategy is used in showing the following theorem.

Theorem 5.2. *The locally state fair nontermination problem for Petri nets is Σ_1-hard.*

Proof. We use a reduction from the halting problem for 2-counter machines, which is known to be Σ_1-hard [26]. Let M be an arbitrary 2-counter machine of [26]. We will construct a Petri net $\mathscr{C}$ and a partition $\mathscr{T}$ of $\mathscr{C}$'s transitions such that $\mathscr{C}$ has a locally state fair nonterminating computation with respect to $\mathscr{T}$ iff M accepts ε (the empty string). Without loss of generality, assume M is deterministic and has a unique final state q_f that can only be entered when both counters are 0. The strategy is similar to that given by Carstensen [3]. First, an arbitrary natural number is generated. This number is then used to bound the length of the simulation and the size of the counters. Accepting computations are then repeatedly simulated so that eventually all zero tests are correct.

The Petri net $\mathscr{C}$ is divided into two main parts, INIT and SIM. $\mathscr{C}$ as well as the partition $\mathscr{T}$ is portrayed in Fig. 5. The purpose of INIT is to generate an arbitrary natural number. In any locally state fair computation, s_2 must eventually fire, permanently disabling s_1. When s_2 fires, SIM may begin simulating a computation of M. At this point, the places *clock*, $\bar{c}_1$, and $\bar{c}_2$ all contain some arbitrary integer n generated by INIT. From this point on, the pairs $(clock, \overline{clock})$, $(c_1, \bar{c}_1)$, and $(c_2, \bar{c}_2)$ are duals in the sense that $clock + \overline{clock} = c_1 + \bar{c}_1 = c_2 + \bar{c}_2 = n$. The transitions $t_1, \ldots, t_m$ then simulate the moves of M by keeping a token in the current state, updating c_1 and c_2 to store the contents of the counters, updating $\bar{c}_1$ and $\bar{c}_2$ so that $c_1 + \bar{c}_1 = c_2 + \bar{c}_2 = n$, decrementing *clock*, and incrementing $\overline{clock}$ each move. Thus, any computation of length n or less may be simulated. If this computation terminates in the accepting



Partitions are denoted by shaded boxes.

Fig. 5. A Petri net for simulating a 2-counter machine.

state q_f, a new computation of length n or less may be simulated. However, there is no guarantee that SIM will always correctly simulate some computation of M. In particular, a move that can only be executed by M when some counter is zero may be executed by SIM when that counter is positive. To overcome this problem, SIM contains two subcomponents, ZERO_1 and ZERO_2. As SIM simulates M entering any state in which a zero-test occurs on some counter, say c_1, it places a token on *query* in ZERO_1, and waits for ZERO_1 to respond by placing a token on either *yes* or *no*. If $c_1 = 0$, $s_6 s_7$ must be fired, so a token is placed on *yes*. If $c_1 \neq 0$, either s_5 or s_6 may be fired. If s_5 fires, a token is placed on *no*. On the other hand, if s_6 fires, s_8 is enabled. Since c_1 never exceeds n, if s_8 is enabled infinitely many times in any locally state fair computation, it must eventually fire, leaving no transitions enabled. Thus, in any locally state fair nonterminating computation, s_6 will fire when $c_1 \neq 0$ at most finitely many times. After the last of these times, SIM correctly simulates moves of M. After this point, q_f can only be entered when both c_1 and c_2 are zero. Thus, in any locally state fair nonterminating computation, $\mathscr{C}$ must eventually simulate correctly an accepting computation of M.

Now suppose M has an accepting computation σ of length n. Let σ' be the computation of $\mathscr{C}$ in which INIT generates n and SIM repeatedly simulates σ. We will now show that σ' is locally state fair with respect to $\mathscr{T}$ by considering each $T_i \in \mathscr{T}$ separately.

- $T_i = \{s_1\}$: s_1 is enabled only finitely many times in σ'.
- $T_i = \{s_2\}$: s_2 is enabled only finitely many times in σ'.
- $T_i = \{s_3\}$: s_3 is fired every time it is enabled in σ'.
- $T_i = \{s_4\}$: $\ddot{T}_i = \{q_0, q_f\}$, all markings reached in σ' at which s_4 is enabled have $q_0 = 0$ and $q_f = 1$, and s_4 is fired infinitely often.
- $T_i = \{s_5\}$: s_5 is fired every time it is enabled in σ'.
- $T_i = \{s_6, s_7\}$: $\ddot{T}_i = \{query, test, yes\}$, and all markings reached in σ' at which s_6 is enabled have $query = 1$, $test = 0$, and $yes = 0$. Since σ ends with both counters zero, we can assume without loss of generality that σ contains at least one successful zero-test; hence, s_6 is fired infinitely often in σ'. s_7 is fired every time it is enabled in σ'.
- $T_i = \{t_1, \ldots, t_m\}$: Since M is deterministic, each of these transitions is fired every time it is enabled in σ'.

We can conclude that $\mathscr{C}$ has a locally state fair nonterminating computation with respect to $\mathscr{T}$ iff M has an accepting computation on ε. $\square$

The above proof shows how local state fairness can be used to force the generation of arbitrary natural numbers and zero-testing. In [16], we showed how these capabilities can typically be used to show fair nontermination problems for Petri nets to be Σ_1^1-complete. However, such is not the case with respect to local state fairness, because each locally state fair computation must be bounded in order for zero-testing to be enforced. The next two lemmas and the subsequent theorem show the locally state fair nontermination problem to be in Σ_1.

Lemma 5.3. *Let p be a place in a Petri net $\mathscr{C}=(P, T, \varphi, \mu_0)$. If $\sigma=\mu_0\mu_1\ldots$ is an infinite computation in which p is unbounded, then σ fires a sequence θ such that $\mu_i \xrightarrow{\theta} \mu_j$, $\mu_i<\mu_j$, $\mu_i(p)<\mu_j(p)$, and if p' is any place that is bounded in σ, $\mu_i(p')=\mu_j(p')$.*

Proof. Since p is unbounded, there is an infinite sequence I of natural numbers $i_0<i_1<\cdots$ such that $\mu_{i_0}(p)<\mu_{i_1}(p)<\cdots$. Furthermore, there must be an infinite subsequence I' of I, $i'_0<i'_1<\cdots$ such that $\mu_{i'_0}<\mu_{i'_1}<\cdots$. Clearly, there must exist $i,j\in I'$ such that $i<j$ and for all places p' that are bounded in σ, $\mu_i(p')=\mu_j(p')$. □

Lemma 5.4. *Let $\mathscr{C}=(P, T, \varphi, \mu_0)$ be a Petri net, and let $\mathscr{T}$ be a partition of T. If there is a locally state fair nonterminating computation of $\mathscr{C}$ with respect to $\mathscr{T}$, then there exist finite firing sequences θ_1 and θ_2 such that $\mu_0 \xrightarrow{\theta_1} \mu_1 \xrightarrow{\theta_2} \mu_2 \xrightarrow{\theta_2} \mu_3 \xrightarrow{\theta_2} \cdots$ is a locally state fair computation of $\mathscr{C}$ with respect to $\mathscr{T}$.*

Proof. Let σ be a locally state fair nonterminating computation of $\mathscr{C}$ with respect to $\mathscr{T}$. Let U be the set of places unbounded in σ, and let B be the set of places bounded in σ. Let $\sigma_1=\mu_0 \xrightarrow{\theta_1} \mu_1$ be a finite prefix of σ containing all markings μ such that the submarking $\mu(B)$ is reached only finitely many times in σ, and such that the submarking $\mu_1(B)$ is reached infinitely many times in σ. Let σ_2 be the remainder of σ. For each $p\in U$, let θ_p be the firing sequence from σ_2 guaranteed by Lemma 5.3. Let $\sigma_3=\mu_1 \xrightarrow{\theta} \mu_2$ be a finite prefix of σ_2 such that

(1) θ contains all θ_p such that $p\in U$;

(2) if the transition t is fired from infinitely many markings in σ_2 containing the submarking $\mu(B)$, then t is fired from some marking μ' in σ_3 such that $\mu(B)=\mu'(B)$ (since there are only finitely many distinct values of $\mu(B)$ such that μ is in σ_2, this condition is satisfied by a finite prefix of σ_2); and

(3) $\mu_2(B)=\mu_1(B)$ (since $\mu_1(B)$ is reached infinitely many times in σ, this condition can be satisfied).

We now construct θ_2 from θ as follows. Scanning θ from left to right, when the beginning of a loop θ_p is encountered, insert enough copies of θ_p to make the displacement of p in the resulting firing sequence positive. Since the displacement of each θ_p is positive, the resulting sequence θ_2 has a positive displacement. Furthermore, since each θ_p has a zero displacement on all places in B, and since $\mu_1(B)=\mu_2(B)$, θ_2 has a zero displacement on all places in B. Hence, only submarkings wholly contained in B are repeated infinitely often in $\mu_0 \xrightarrow{\theta_1} \mu_1 \xrightarrow{\theta_2} \mu_2 \xrightarrow{\theta_2} \mu_3 \xrightarrow{\theta_2} \cdots$. Thus, from condition (2) above, θ_1 and θ_2 satisfy the lemma. □

Theorem 5.5. *The locally state fair nontermination problem for Petri nets is in Σ_1.*

Proof. Let $\mathscr{C}=(P, T, \varphi, \mu_0)$ be an arbitrary Petri net, and let $\mathscr{T}$ be an arbitrary partition of T. We now describe a TM M that accepts $(\mathscr{C}, \mathscr{T})$ iff $\mathscr{C}$ has a locally state fair nonterminating computation with respect to $\mathscr{T}$. (Note that M is not required to halt on all inputs.) M first guesses θ_1 and θ_2 given by Lemma 5.4, and verifies that $\mu_0 \xrightarrow{\theta_1} \mu_1 \xrightarrow{\theta_2} \mu_2$ such that $\mu_1 \leq \mu_2$. M then verifies that $\sigma = \mu_0 \xrightarrow{\theta_1} \mu_1 \xrightarrow{\theta_2} \mu_2 \xrightarrow{\theta_2} \mu_3 \xrightarrow{\theta_2} \cdots$ is locally state fair with respect to $\mathcal{T}$. This verification is done in the following manner. Let $B = \{p \mid \mu_1(p) = \mu_2(p)\}$, and let $U = \{p \mid \mu_1(p) < \mu_2(p)\}$. Clearly, if $\ddot{T}_i$ contains a place in U, there will be only finitely many markings μ in σ such that $\mu(\ddot{T}_i) = \mu'(\ddot{T}_i)$ for any given μ'. Hence, M only needs to consider those elements $T_i \in \mathcal{T}$ such that $\ddot{T}_i \subseteq B$. M therefore verifies that for all μ in $\mu_1 \xrightarrow{\theta_2} \mu_2$, if transition $t \in T_i$ such that $\ddot{T}_i \subseteq B$ is enabled at μ, then t is fired at some μ' in $\mu_1 \xrightarrow{\theta_2} \mu_2$ such that $\mu(\ddot{T}_i) = \mu'(\ddot{T}_i)$. Once this is verified, M accepts. From Lemma 5.4, M accepts iff $\mathcal{C}$ has a locally state fair nonterminating computation with respect to $\mathcal{T}$. $\square$

The following result follows immediately from Theorems 5.2 and 5.5.

Corollary 5.6. *The locally state fair nontermination problem for Petri nets is Σ_1-complete.*

As mentioned above, local state fairness with respect to Petri nets cannot be used to enforce arbitrary use of both zero-testing and the generation of arbitrary natural numbers. However, concurrent programs can perform zero-testing without relying on a fairness constraint. Therefore, we can use local state fairness with concurrent programs to generate arbitrary natural numbers at will. This capability allows us to show that the locally state fair nontermination problem for concurrent programs is Σ_1^1-complete. In order to characterize the class Σ_1^1, we introduce the notion of *infinite-branching programs.* An infinite-branching program is simply a program with the added ability to nondeterministically generate an arbitrary natural number. We now give the following lemma from [5], which gives a characterization of Σ_1^1.

Lemma 5.7 (from [5]). *The set of all infinite-branching programs that contain an infinite computation is Σ_1^1-complete.*

We can now show the following theorem.

Theorem 5.8. *The locally state fair nontermination problem for concurrent programs is Σ_1^1-complete.*

Proof. We will show this problem to be equivalent to determining whether an infinite-branching program contains an infinite computation. Let P be an infinite-branching program. We will show how to construct a system (P_1, P_2) of programs (without infinite-branching) that has a locally state fair nonterminating computation iff P has an infinite computation. First, consider Table 4. Suppose Program P_2 is completely deterministic. (Note that the portion shown is.) Since both P_1 and P_2 are deterministic, any nonterminating computation in which both programs execute transitions infinitely often is locally state fair. Furthermore, it is easily verified that any nonterminating computation in which P_1 executes transitions only finitely many

Table 4
Simulation of infinite-branching.

Program P_1	**Program P_2**
l_1: **if** true **then** $x := 1$ **goto** l_1	...
	m_1: **if** true **then** $n := 0$; $x := 0$ **goto** m_2
	m_2: **if** $x = 1$ **then goto** m_3
	□
	if $x = 0$ **then** $n := n + 1$ **goto** m_2
	m_3: ...

times is not locally state fair. Thus, in a locally state fair computation, n may be incremented to an arbitrary value, as long as the resulting computation is nonterminating, but the loop at m_2 must always terminate. Since this technique can be used to simulate bounded nondeterminism as well as infinite branching, we can construct a deterministic program P_2 such that (P_1, P_2) has a locally state fair nonterminating computation iff P has an infinite computation.

Now let $S = (P_1, \ldots, P_k)$ be an arbitrary system of k concurrent programs (without infinite branching). We will construct a program P with infinite-branching that has an infinite computation iff S has a locally state fair nonterminating computation. P simulates S by nondeterministically selecting transitions to simulate. As the simulation progresses, P maintains a table containing all local states reached by each P_i. When a new local state q is reached by some P_i, P guesses for each transition t of P_i enabled at q how many times P_i will be in state q before t is executed. P then stores this value as $n_{q,t}$. Each time P_i is in state q, P decrements $n_{q,t}$. Each time t is executed from state q, P guesses a new $n_{q,t}$. If, after a new $n_{q,t}$ is generated, some $n_{q',t'}$ is zero, P halts. Clearly, P has an infinite computation iff S has a locally state fair nonterminating computation. □

As our last result, we show that the globally state fair nontermination problem for concurrent programs is Π_1-complete. In order to show this, we must describe a TM that can verify that there is *no* globally state fair nonterminating computation in a given system. The main reason we are able to do this is given in the following lemma.

Lemma 5.9. *Any system S of concurrent programs having an infinite set of reachable states has a globally state fair nonterminating computation.*

Proof. Consider the reachability tree T of S defined as follows. Let the initial state of S be the root of T. For each nonterminal state q in T, the children of q are all states reachable from q by one move of S, except those states that are ancestors of q (where q is considered an ancestor of itself). Since the set of reachable states is infinite, T must be infinite. Since T has a bounded branching factor, T must have infinite depth. Therefore, there is an infinite path in T representing a nonterminating computation in which no state is entered more than once. This nonterminating computation is globally state fair. □

Theorem 5.10. *The globally state fair nontermination problem for concurrent programs is Π_1-complete.*

Proof. Given an arbitrary deterministic TM M, we can clearly construct a deterministic program P that terminates iff M halts on ε. Since the only computation of a single deterministic program is always globally state fair, P has a globally state fair nonterminating computation iff M does not halt. The problem is therefore Π_1-hard.

Let S be a system of concurrent programs. We will describe a TM M that accepts S iff S has no globally state fair nonterminating computation. M first begins to construct G_S. If this graph is infinite M will never accept S, but from Lemma 5.9 there is a globally state fair nonterminating computation. If the graph is finite, when M completes the graph it then searches the graph for a g-knot, accepting iff one does not exist. From Lemma 4.3, M accepts S iff S has no globally state fair nonterminating computation. Therefore, the problem is Π_1-complete. □

Acknowledgment

We would like to thank the referee for suggestions which improved the content as well as the presentation of our results.

References

[1] E. Best, Fairness and conspiracies, *Inform. Process. Lett.* **18** (1984) 215-220; Addendum, *ibidem* **19** (1984) 162.
[2] G. Bochmann and C. Sunshine, Formal methods in communication protocol design, *IEEE Trans. Commun.* (1980) 624-631.
[3] H. Carstensen, Decidability questions for fairness in Petri nets, in: *Proc. 4th Ann. Symp. on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **247** (Springer, Berlin, 1987) 396-407.
[4] H. Carstensen and R. Valk, Infinite behaviour and fairness in Petri nets, in: *Advances in Petri Nets 1984*, Lecture Notes in Computer Science **188** (Springer, Berlin, 1985) 83-100.
[5] A. Chandra, Computable nondeterministic functions, in: *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978) 127-131.
[6] A. Chandra, D. Kozen and L. Stockmeyer, Alternation, *J. ACM* **28** (1) (1981) 114-133.
[7] E. Clarke and E. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: *Workshop on Logics of Programs*, Lecture Notes in Computer Science **131** (Springer, Berlin, 1981) 52-71.
[8] E. Emerson and C. Lei, Modalities for model checking: branching time logic strikes back, *Sci. Comput. Programming* **8** (1987) 275-306.
[9] R. Fagin, J. Halpern and M. Vardi, A model-theoretical analysis of knowledge, in: *Proc. 25th IEEE Symp. on Foundations of Computer Science* (1984) 268-278.
[10] M. Gouda and C. Chang, Proving liveness for networks of communicating finite state machines, *ACM Trans. on Programming Languages and Systems* **8** (1) (1986) 154-182.
[11] M. Hack, Analysis of production schemata by Petri nets, MAC TR-94, Project MAC, MIT, Cambridge, MA, 1972.
[12] J. Halpern and Y. Moses, Knowledge and common knowledge in a distributed environment, in: *Proc. 3rd ACM Symp. on Principles of Distributed Computing* (1984) 50-61.

[13] S. Hart, M. Sharir and A. Pnueli, Termination of probabilistic concurrent programs, *ACM Trans. on Programming Languages and Systems* **5** (3) (1983) 356-380.

[14] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978) 666-677.

[15] R. Howell and L. Rosier, Problems concerning fairness and temporal logic for conflict-free Petri nets, *Theoret. Comput. Sci.* **64** (1989) 305-329.

[16] R. Howell, L. Rosier and H. Yen, A taxonomy of fairness and temporal logic problems for Petri nets, to appear in *Theoret. Comput. Sci.*

[17] N. Jones, L. Landweber and E. Lien, Complexity of some problems in Petri nets, *Theoret. Comput. Sci.* **4** (1977) 277-299.

[18] P. Kanellakis and S. Smolka, On the analysis of cooperation and antagonism in networks of communicating processes, in: *Proc. 4th ACM Symp. on Principles of Distributed Computing* (1985) 23-38.

[19] R. Karp and R. Miller, Parallel program schemata, *J. Comput. System Sci.* **3** (1969) 167-195.

[20] R. Ladner, The complexity of problems in systems of communicating sequential processes, *J. Comput. System Sci.* **21** (1980) 179-194.

[21] D. Lehmann, Knowledge, common knowledge and related puzzles, in: *Proc. 3rd ACM Symp. on Principles of Distributed Computing* (1984) 62-67.

[22] D. Lehmann and M. Rabin, On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem, in: *Proc. 10th ACM Symp. on Principles of Programming Languages* (1981) 133-138.

[23] D. Lehmann, A. Pnueli and J. Stavi, Impartiality, justice and fairness: the ethics of concurrent termination, in: *Proc. 8th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science **115** (Springer, Berlin, 1981) 264-277.

[24] R. Lipton, The reachability problem requires exponential space, Tech. Rept. 62, Dept. of Computer Science, Yale University, 1976.

[25] Z. Manna and A. Pnueli, How to cook a temporal proof system for your pet language, in: *Proc. 10th Ann. ACM Symp. on Principles of Programming Languages* (1983) 141-154.

[26] M. Minsky, *Computation: Finite and Infinite Machines* (Prentice Hall, Englewood Cliffs, NJ, 1967).

[27] J. Peterson, *Petri Net Theory and the Modeling of Systems* (Prentice Hall, Englewood Cliffs, NJ, 1981).

[28] A. Pnueli, On the extremely fair treatment of probabilistic algorithms, in: *Proc. 15th Ann. ACM Symp. on Theory of Computing* (1983) 278-290.

[29] J. Queille and J. Sifakis, Fairness and related properties in transition systems—a temporal logic to deal with fairness, *Acta Inform.* **19** (1983) 195-220.

[30] M. Rabin, The choice coordination problem, *Acta Inform.* **17** (1982) 121-134.

[31] M. Rabin, N-process synchronization by $4 * \log_2 n$-valued shared variable, in: *Proc. 21st Ann. Symp. on Foundations of Computer Science* (1980) 407-410.

[32] C. Rackoff, The covering and boundedness problems for Vector Addition Systems, *Theoret. Comput. Sci.* **6** (1978) 223-231.

[33] J. Reif, Universal games of incomplete information, in: *Proc. 11th ACM Symp. on Theory of Computing* (1979) 288-308.

[34] W. Reisig, *Petri Nets—An Introduction* (Springer, Berlin, 1985).

[35] H. Rogers, *Theory of Recursive Functions and Effective Computability* (The MIT Press, Cambridge, MA, 1987).

[36] L. Rosier and H. Yen, On the complexity of deciding fair termination of probabilistic concurrent finite-state programs, *Theoret. Comput. Sci.* **58** (1988) 263-324.

[37] A. Sistla and S. German, Reasoning with many processes, in: *Proc. IEEE Symp. on Logic in Computer Science* (1987) 138-152.

[38] A. Sistla and E. Clarke, The complexity of propositional linear temporal logic, *J. ACM* **32** (1985) 733-749.

[39] M. Vardi, Automatic verification of probabilistic concurrent finite-state programs, in: *Proc. 26th Ann. Symp. on Foundations of Computer Science* (1985) 327-338.