Information and Computation **166**, 1–34 (2001)
doi:10.1006/inco.2000.3014, available online at http://www.idealibrary.com on IDEAL®

On Fairness Notions in Distributed Systems

I. A Characterization of Implementability*

Yuh-Jzer Joung†

Department of Information Management, National Taiwan University, Taipei, Taiwan
E-mail: joung@ccms.ntu.edu.tw

Received January 28, 1998

This is the first part of a two-part paper in which we discuss the implementability of *fairness notions* in distributed systems where asynchronous processes interact via synchronous constructs—usually called *multiparty interactions*. In this part we present a criterion for fairness notions and show that if a fairness notion violates the criterion, then no deterministic algorithm for scheduling multiparty interactions can satisfy the fairness notion. Conversely, the implementation is possible if the criterion is obeyed. Thus, the criterion is sufficient and necessary to guarantee the implementability of all possible fairness notions. To our knowledge, this is the first such criterion to appear in the literature. The main benefit of the proposed criterion is that it reduces reasoning about a complex and concrete implementation model to reasoning about a simpler and abstract model for process interaction. To illustrate this, we use the criterion to examine several important fairness notions, including *strong interaction fairness*, *strong process fairness*, *weak process fairness*, *U-fairness*, and *hyperfairness*. All, except weak process fairness, fail to pass the criterion. Moreover, we also apply the criterion to analyze the system structures rendering the impossibility phenomena. This analysis helps us separate, for each fairness notion, the set of systems for which the fairness notion can be implemented from those for which it cannot.

INTRODUCTION

Since Hoare introduced CSP [22], *interactions* and *nondeterminism* have become two fundamental features in many high-level programming languages for distributed computing and algebraic models of concurrency, e.g., Ada [48], Occam [23], CCS [38], and π-calculus [39]. Interactions serve as a synchronization and communication mechanism: the participating processes of an interaction must synchronize before embarking on any data transmission. Nondeterminism allows a process to choose from a set of potential interactions it has specified one interaction to execute.

Note that interactions in CSP and Ada can involve only two processes. However, more recent language developments (e.g., SCCS [37], CIRCAL [36], Script [19], Compact [15], Action Systems [7], SR [3,16], LOTOS [11], Extended LOTOS [13], IP [17], and DisCo [24, 25]) have extended these biparty activities to a more general case, *multiparty interactions*, allowing an arbitrary number of processes to interact. More precisely, a multiparty interaction is a synchronous action involving a fixed set of participant processes. An attempt to execute the action by a participant process delays the process until all other participants are ready to execute the action. After the execution each participant process continues its local computation. It is believed that multiparty interactions provide a higher level of abstraction and encourage modular programming and design [17, 18, 32]. For example, the natural unit of process interaction in the famous Dining Philosophers problem involves a philosopher and its two neighboring chopsticks; that is, a three-party interaction.

The implementation of multiparty interactions is concerned with synchronizing asynchronous processes to participate in interactions so that the following two requirements are satisfied:

(1) *Synchronization.* If a process starts to execute an interaction, then all other participants of the interaction will also execute the interaction.

* A preliminary version of this paper appeared as Characterizing fairness implementability for multiparty interaction, *in* "Proceedings of the 23rd International Colloquium on Automata, Languages and Programming, Paderborn, Germany, July 8–12," Lecture Notes in Computer Science, Vol. 1099, pp. 110–121, Springer, Berlin, 1996. This research was supported by the National Science Council, Taipei, Taiwan, under Grants NSC 85-2213-E-002-059 and NSC 86-2213-E-002-053.

† The author is currently visiting Laboratory for Computer Science, Massachusetts Institute of Technology (1999–2000).

0890-5401/01 $35.00

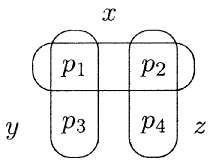

FIG. 1. A system of four processes p_1, p_2, p_3, and p_4, and three interactions x, y, and z.

(2) *Mutual exclusion.* Conflicting interactions are not executed simultaneously, where two interactions *conflict* if they involve a common member process.

Because nondeterminism allows a process to choose from a set of potential interactions an arbitrary interaction to execute, with an improper interaction scheduling, an implementation of multiparty interactions may render an undesirable program behavior, usually because it violates some liveness property. So some fairness notion is typically imposed on the problem to exclude unwanted computations that would otherwise be legal.

To illustrate, consider a system of four processes p_1, p_2, p_3, and p_4 and three interactions x, y, and z, where x involves the set of processes $\{p_1, p_2\}$, y involves $\{p_1, p_3\}$, and z involves $\{p_2, p_4\}$ (see Fig. 1). Assume that each process p_i, $1 \leq i \leq 4$, transits between an *idle* state where it is busy in its local actions and a *ready* state where it wishes to establish some interaction of which it is a member. Let

$$(p_1 p_3 y p_2 p_4 z)^\omega$$

denote a repeated scenario in which p_1 and p_3 become ready and jointly execute y, and then p_2 and p_4 become ready and execute z. The computation then satisfies *strong interaction fairness* (SIF), meaning that an interaction that is infinitely often *enabled* (that is, with its participants all ready) is executed infinitely often. The computation

$$(p_1 p_3 p_2 y p_4 z)^\omega$$

does not satisfy SIF because x is enabled in every state immediately after p_2 is ready but it is never executed.

A fairness notion is said *implementable* for a system if there is an implementation of multiparty interactions such that all computations of the system satisfy the fairness notion. We focus here on the implementability of fairness notions in distributed systems where asynchronous processes interact via multiparty interactions. In a companion paper [28] we compose several hierarchies of fairness notions in terms of their expressiveness, and for each of the hierarchies we delineate the line between implementable and unimplementable fairness notions.

1 APPRAISING FAIRNESS NOTIONS

Since a fairness notion excludes from all possible computations some that would otherwise be valid, in general, any subset of computations could be considered as a semantic constraint for the system. However, not many of them are useful, and so criteria have been proposed for determining their appropriateness, including the following [4]:

Feasibility: Every partial computation can be extended to a valid one.[1]

Equivalence-robustness: Equivalent computations are either all valid, or all invalid. Computations are *equivalent* if they are identical up to the order of independent actions. Here we assume that the underlying semantics induces a *dependency* relation on actions of the system, which is usually a partial order reflecting Lamport's causality relation [34].

Feasibility is often demonstrated by an explicit scheduler, which proceeds in lock-step synchrony with the system, and has complete knowledge of the global state of the system at all times. In each step

[1] The notion of feasibility is also equivalent to Abadi and Lamport's *machine closure* [1, 2].

the scheduler determines for the system an action to execute and waits until the execution terminates before it moves on to the next step. Note that all actions, local and non-local (i.e., interactions), are scheduled by the scheduler. A fairness notion can then be proved to be feasible by exhibiting such a scheduler so that every partial computation can be generated by the scheduler, and every complete computation generated by the scheduler is valid.

Note that because in a distributed environment no process can have a complete knowledge of the global state of the system at all times, an explicit scheduler does not directly correspond to a real implementation. However, using a technique of "superimposition" [4, 12] one can convert a scheduler into a real scheduling program executed in parallel with the main program (i.e., the one that the system is executing). In each step the scheduler communicates with every process in the system to obtain the global state information. The scheduler then determines the next action for the system, informs every process that is responsible for the execution of the selected action to execute the action, and then waits for the execution to terminate before it proceeds to the next step. All other processes' executions of the main program are suspended until further notice by the scheduler. (This can be done, for example, by augmenting each action with a Boolean variable to enable/disable the action.)

For the sake of efficiency, however, most practical implementations for biparty and multiparty interactions (e.g., [9, 10, 14, 21, 26, 29, 33, 43, 45, 47]) do not use the aforementioned superimposition technique. Rather, they allow processes to execute local actions on their own. The scheduling takes place only when some process is ready for interaction, and only interactions are scheduled. More importantly, unlike the superimposition technique, the implementations do not depend on whether local actions and interactions terminate or not (that is, whether a process will eventually become ready for interaction). Note that superimposition may result in a deadlock if the action the scheduling program is waiting for does not terminate while some other interaction is enabled for execution. As a result, most implementations for multiparty interactions make use of the assumption that processes decide autonomously when they will be ready for interaction.

It turns out that if processes can decide autonomously when they will be ready for interaction, then feasibility alone does not necessarily guarantee implementability. To illustrate, the notion of SIF is feasible [4], but its implementation has been proven impossible by any deterministic algorithm [26, 46].

On the other hand, feasibility is not a necessary condition for implementation either, regardless of whether local actions and interactions terminate or not. To see this, consider a system of two processes p_1 and p_2, and two interactions x and y, both involving p_1 and p_2. Assume again that each process transits between an idle state and a ready state, where in the ready state every process is ready for both x and y. Let $\mathbb{C}$ be a fairness notion that prohibits y from being executed. Clearly, $\mathbb{C}$ can be implemented for the system by always letting the two processes execute x whenever they are ready for interaction. However, $\mathbb{C}$ is not feasible because $p_1 p_2 y$ (which represents that p_1 and p_2 become ready and then jointly execute y) is a partial computation of the system, but it cannot be extended to a complete computation satisfying $\mathbb{C}$.

For the equivalence-robustness criterion, it is observed that most equivalence-robust fairness notions are implementable. (See [4] for a reference of such fairness notions.) This holds even if the time when a process will be ready for interaction cannot be determined in advance. As we shall see in Section 4, the observation is not coincidental because under a notion of "strong feasibility" equivalence-robustness suffices to guarantee implementability. Equivalence-robustness, however, is not necessary for every implementable fairness notion. For example, consider the notion of *weak process fairness* (WPF), which requires a process continually ready for an enabled interaction (not necessarily the same one, though) to execute some interaction eventually. WPF is not equivalence-robust [4], but it can be implemented in a system consisting of only biparty interactions [26, 47]. (In fact, WPF is also possible for multiparty interaction. See Section 5.3)

In this paper we propose a new criterion for appraising fairness notions. The criterion requires that a fairness notion be realized by an *abstract scheduling function* such that all computations produced by this function are valid (with respect to the fairness notion), and all other computations *indistinguishable* from the produced computations are also valid. Intuitively, the abstract function captures the scheduling policy adopted by a concrete scheduling program, while the indistinguishableness relation expresses properties of computations that cannot be distinguished by any asynchronous distributed environment.

Assume the following in the underlying model of computation:

A1. One process's readiness for multiparty interaction can be known by another only through communication, and the time it takes two processes to communicate is nonnegligible.

A2. A process decides autonomously when it will attempt an interaction, and at a time that cannot be predicted in advance.

We show that if a fairness notion violates the criterion, then no deterministic algorithm for multiparty interaction scheduling can satisfy the fairness notion. For fairness notions that satisfy the criterion, we also present a general algorithm to implement them in an asynchronous system where processes communicate exclusively by biparty message passing. Thus, the criterion is sufficient and necessary to determine the implementability of any given fairness notion. To our knowledge, this is the first such criterion to appear in the literature.

The main benefit of the proposed criterion is that it reduces reasoning about a complex and concrete implementation model to reasoning about a simpler and abstract model for process interaction. To illustrate this, we use the criterion to examine several important fairness notions, including SIF, *strong process fairness* (SPF) [4], *U-fairness* [5], and *hyperfairness* [6]. We also apply the criterion to analyze the system structures rendering the impossibility phenomena. This analysis helps us separate, for each given fairness notion, the set of systems for which the fairness notion can be implemented from those for which it cannot.

The rest of the paper is organized as follows. Section 2 presents an abstract model for process interaction and an implementation model for interaction scheduling. The relation between the two models is also described. Section 3 presents our criterion and shows that it is necessary and sufficient to determine the implementability of any given fairness notion. Section 4 exploits properties of implementable fairness notions derived from the criterion. In Section 5 we use the criterion to examine several fairness notions that are commonly associated with multiparty interactions. Section 6 discusses related work and then concludes.

2 PRELIMINARIES

21. An Abstract Model for Process Interaction

An *interaction system* is a triple $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$, where P is a finite set of processes, I is a finite set of interactions, and M is a program. Each interaction x involves a fixed set $P_x \subseteq \mathsf{P}$ of participant processes, and can be executed by the participants (and only the participants) only if they are all ready for the interaction. A process is either in an *idle* state or in a *ready* state. Initially, all processes are idle. An idle process p may autonomously become ready, where it is ready for a set $p.aim$ of potential interactions of which it is a member. After executing one interaction in $p.aim$, p returns to an idle state; see Fig. 2. Set $p.aim$ is determined by M based on the history of interactions p has executed. On some occasions we may consider programs allowing a process to be ready for all interactions of which it is a member every time when the process is ready for interaction. We use $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^{\forall})$ to denote an interaction system associated with this type of programs.

A state s of $\mathbb{IS}$ consists of the history of interactions the system has executed so far, and for each $p \in \mathsf{P}$, the state (i.e., idle or ready) of p and the set of potential interactions p are ready to execute when p is in a ready state. We use $[s]_{hist}$ to denote the history of s, $[s]_p$ the state of p in s, and $[s]_{p.aim}$ the set of potential interactions p is ready to execute. We assume that $[s]_{p.aim} = \emptyset$ if $[s]_p = idle$. Moreover,

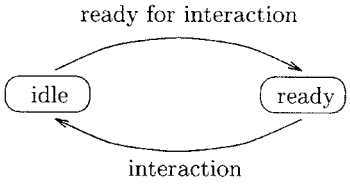

FIG. 2. The state transition diagram of a process.

$[s]_{hist,p}$ denotes the sequence of interactions in $[s]_{hist}$ that involve p, i.e., the history of interactions executed by p. An interaction x is *enabled* in s iff every process $p \in P_x$ is ready for x, i.e., $[s]_p = ready$ and $x \in [s]_{p.aim}$. Let S be the set of all possible states of $\mathbb{IS}$. State transitions are written as $s \xrightarrow{a} s'$, where $s, s' \in \mathsf{S}$, and a is the action whose execution results in the transition. State transitions are of the following forms:

Ready: $s \xrightarrow{p.I} s'$ iff $[s]_{hist} = [s']_{hist}$, $[s]_p = idle$, $[s']_p = ready$, $\mathsf{M}(p, [s]_{hist,p}) = [s']_{p.aim} = I$, and $\forall q \in \mathsf{P} - \{p\}$, $[s]_q = [s']_q$ and $[s]_{q.aim} = [s']_{q.aim}$.

That is, the action $p.I$ transits process p from idle to a state ready for the set I of interactions.

Interaction: $s \xrightarrow{x} s'$ iff $[s']_{hist} = [s]_{hist} \cdot x$, $\forall p \in P_x, [s]_p = ready$ and $x \in [s]_{p.aim}$ and $[s']_p = idle$ and $[s']_{p.aim} = \emptyset$, and $\forall q \in \mathsf{P} - P_x$, $[s]_q = [s']_q$ and $[s]_{q.aim} = [s']_{q.aim}$.

That is, the execution of interaction x transits all participants of x from state ready to idle.

A *run* π is a sequence of the form

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots,$$

where s_0 is the initial state (that is, $[s_0]_{hist} = \epsilon$ and $\forall p \in \mathsf{P}, [s_0]_p = idle$ and $[s_0]_{p.aim} = \emptyset$), and each $s_i \xrightarrow{a_{i+1}} s_{i+1}$ is a state transition of the system. (In the paper ϵ denotes the empty sequence such that for all finite sequence π, $\pi\epsilon = \epsilon\pi = \pi$.) In particular, π is *complete* if it is infinite or it ends up in a state in which all processes are ready but no interaction is enabled; otherwise, π is *partial*. We use run$^*(\mathbb{IS})$ to denote the set of all finite runs of $\mathbb{IS}$, and run $(\mathbb{IS})$ denotes the set of complete runs. Thus, run$^*(\mathbb{IS}) \cap$ run $(\mathbb{IS})$ is the set of finite complete runs.

Since each run $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$ is uniquely determined by the sequence of actions executed in the run, we often write the run as $a_1 a_2 \ldots$. Conversely, we call a sequence of actions $a_1 a_2 \ldots$ a run if it represents a legal run of $\mathbb{IS}$. It should be noted that when using actions to represent a run, actions are distinguished by their occurrences. For example, the two $p.I$'s in run $p.I\ x\ p.I\ x \ldots$ represent different instances of actions. If necessary, we can use superscripts to distinguish them.

Some comments on our model are given in order. First, by stipulating that p returns to an idle state after interaction, we have also assumed *noninstantaneous readiness* as in [4], which means that a process cannot be immediately ready for interaction after executing some interaction. Thus an interaction cannot be "continuously" enabled throughout an interval if some process involved in the interaction has executed an interaction in the interval.

Second, many languages that use interactions in guards (e.g., CSP) allow a choice between local actions and interactions. That is, a process p ready for interaction may in effect performs some local action and then returns to an idle state without establishing any interaction with other processes. We can model this non-uniform choice by dedicating some local interactions involving only p to p. By including these local interactions in $p.aim$, p can have a choice between local actions and interactions in a ready state.

Third, in our model process termination can be expressed by the ready action $p.\emptyset$; that is, p is no longer willing to engage in any interaction.

Finally, we do not distinguish finite runs which are complete because every process terminates from those which are complete because the system is deadlocked—some processes are ready to execute an interaction but no interaction is enabled. Also, unlike finite complete runs, the definition of infinite runs does not assume "bounded transition time." So in an infinite run a process may stay idle forever, and similarly a set of processes may be ready for an interaction indefinitely. We leave the decision whether such scenarios are allowed or not to be determined explicitly by the underlying fairness notion. Most systems, however, do impose a very weak fairness notion to exclude the above scenarios. Alternatively, such a fairness notion can be incorporated directly into the definition of complete runs so that the bounded transition time assumption is made for both finite and infinite complete runs; for example, see [7]. On the other hand, the bounded transition time can be removed from finite complete runs by introducing a null action λ into the abstract model so that for every state s, $s \xrightarrow{\lambda} s$ (called a stuttering step in [2]).

DEFINITION 2.1. *A fairness notion* $\mathbb{C}$ is a function which, given an interaction system $\mathbb{IS}$, returns a set of complete runs $\mathbb{C}(\mathbb{IS}) \subseteq \text{run}(\mathbb{IS})$. We say that π is $\mathbb{C}$-*valid* ($\mathbb{C}$-*fair*, or simply *valid* or *fair* when the context is clear) if $\pi \in \mathbb{C}(\mathbb{IS})$.

We assume that actions involving a common participant process in a run are totally ordered by the ordering the process executes them (which in turn is induced by the semantics of the underlying program). These total orderings then induce a typical partial order dependency relation on the actions of a run such that $a \prec b$ iff some process executes a before b, or there exists c such that $a \prec c$ and $c \prec b$; see [34]. Two runs π and ρ are *equivalent*, denoted by $\pi \equiv \rho$, iff for every process p, the sequence of actions involving p in π is the same as that in ρ. As can be seen, if $\pi \equiv \rho$, then one of them can be obtained from the other by transpositions of independent actions.

For example, consider the following run of the interaction system shown in Fig. 1, and assume that the system is associated with a program $\mathsf{M}^{\forall}$:

$$\pi = (p_1 p_3 y p_2 p_4 z)^{\omega}.$$

For notational simplicity we overload the notation p_i to abbreviate the action $p_i.I$, where $I = \{x \in \mathsf{I} \mid p_i \in P_x\}$, i.e., the action that process p_i readies all interactions of which it is a member. This abbreviation will be adopted throughout the paper. Observe that every instance of y in π is independent of the following action p_2. So π is equivalent to the run

$$(p_1 p_3 p_2 y p_4 z)^{\omega}.$$

Similarly, π is also equivalent to the run

$$(p_2 p_4 p_1 p_3 y z)^{\omega}.$$

22. An Implementation Model for Interaction Scheduling

We now consider the implementation of multiparty interactions. By this we mean augmenting each process in an interaction system with variables and actions, and possibly introducing auxiliary processes so that each ready process knows when and which interaction to execute.

Formally, a *scheduling program* for an interaction system $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ is a sextuple

$$\mathbb{SP}_{\mathbb{IS}} = (\mathsf{P}, \mathsf{I}, \mathsf{M}, \mathsf{Aux}, \{\mathsf{V}_p : p \in \mathsf{P} \cup \mathsf{Aux}\}, \{\mathsf{A}_p : p \in \mathsf{P} \cup \mathsf{Aux}\}),$$

with three extra components Aux, $\{\mathsf{V}_p : p \in \mathsf{P} \cup \mathsf{Aux}\}$, and $\{\mathsf{A}_p : p \in \mathsf{P} \cup \mathsf{Aux}\}$. Aux is a set of processes (possibly empty) that are added to assist the coordination of interactions. To distinguish Aux from P, we refer to the processes in P as *primary* and those in Aux as *auxiliary*. For each $p \in \mathsf{P} \cup \mathsf{Aux}$, V_p is the set of variables local to p, and A_p is the set of actions executed by p. We assume that processes communicate by reliable, FIFO, biparty asynchronous message passing, although our results in this paper hold as well if communication is by accessing shared variables.

Like the abstract model presented in the previous section, we assume that for each primary process p, V_p contains a variable $p.state$ which designates whether p is idle or ready, a variable $p.aim$ which designates the set of potential interactions p is ready to execute, and a variable $p.hist$ which designates the history of interactions executed by p. Moreover, V_p contains a variable $p.commit$ which designates the interaction p has committed to execution. Variable $p.commit$ is set only once in each ready state and is undefined if p is idle. We assume that p can commit to x only when it is ready for x. Moreover, if some process has committed to x, then all other participants of x will eventually commit to x, and these commitments should not depend on the state of any other primary process not involved in x. When all participants have committed to x, an instance of x is executed and then the participants return to their idle states. Alternatively, $p.commit = x$ may be viewed as that process p has started x.

For every primary process p, actions in A_p can be divided into three types: (1) local actions and/or communications, (2) transitions from idle to ready, and (3) interactions. Local actions and/or

communications are of the form

$$b_\sigma(\mathsf{V}_p);\ \mathit{message\_receptions} \rightarrow\ f_\sigma(\mathsf{V}_p - \{p.\mathit{state},\ p.\mathit{aim},\ p.\mathit{hist}\}),\ \mathit{message\_sendings}, \tag{1}$$

where $b_\sigma(\mathsf{V}_p)$ is a Boolean condition on the variables in V_p, $f_\sigma(V)$ represents the effect of the execution to the variables in V, *message_receptions* express the messages to be received, and *message_sendings* describe the messages to be sent when the action is executed. All four parts are optional. An action can be executed only if it is *enabled*, i.e., $b_\sigma(\mathsf{V}_p)$ evaluates to true and the messages specified in the reception list have arrived. Note that actions of this form respect Assumption A1 (see Section 1) in the sense that a process obtains state information of another only through message passing, and a message's sending and reception must occur in two separate actions; i.e., the communication time is nonnegligible. Note further that this type of actions are not allowed to alter $p.\mathit{state}$, $p.\mathit{aim}$, and $p.\mathit{hist}$.

Transitions from idle to ready are of the form

$$\begin{aligned} p.\mathit{state} = \mathit{idle} \rightarrow\ & p.\mathit{state} := \mathit{ready}, \\ & p.\mathit{aim} := \mathsf{M}(p,\ p.\mathit{hist}), \\ & f_\sigma(\mathsf{V}_p - \{p.\mathit{state},\ p.\mathit{aim},\ p.\mathit{hist}\}), \\ & \mathit{message\_sendings}. \end{aligned} \tag{2}$$

This complies with Assumption A2 in that a process may enter a ready state any time when it is idle. Like form (1), an action of this form may update local variables and send out messages to other processes, possibly to inform them of the process's readiness.

To represent interactions (and state transitions from ready to idle), for each $x \in \mathsf{I}$ of which p is a participant, A_p contains an action of the form

$$\begin{aligned} p.\mathit{commit} = x \rightarrow\ & p.\mathit{state} := \mathit{idle}, \\ & p.\mathit{aim} := \emptyset, \\ & p.\mathit{hist} := p.\mathit{hist} \cdot x, \\ & p.\mathit{commit} := \perp, \\ & f_\sigma(\mathsf{V}_p - \{p.\mathit{state},\ p.\mathit{aim},\ p.\mathit{hist},\ p.\mathit{commit}\}), \\ & \mathit{message\_sendings}. \end{aligned} \tag{3}$$

Recall that $p.\mathit{commit} = x$ only when p is ready for x (i.e., $x \in p.\mathit{aim}$), and that when $p.\mathit{commit} = x$, every other participant of x will eventually set their commit to x. So when some process has set its commit to x, all participants of x will eventually execute their actions of this form to establish an interaction. To simplify the implementation model, we assume that the actions are executed simultaneously by the participants of x.[2] For otherwise, extra variables are needed to prevent a participant p of x from "out-running" other participants in executing instances of x; i.e., to prevent p from committing to x (and then executing x) several times before the other participants of x have committed to (a particular instance of) x.

Since auxiliary processes are added only to assist coordination, they have only actions of form (1). A typical centralized scheduling algorithm, for example, might employ an auxiliary process to collect state information from primary processes, and to direct them to commit to an interaction it has chosen.

As usual, a state α of $\mathbb{SP}_{\mathbb{IS}}$ consists of the values of all variables of the program and the set of messages that have been sent but have not yet been received. A *computation* Π of $\mathbb{SP}_{\mathbb{IS}}$ is a sequence of the form

$$\alpha_0 \xrightarrow{\sigma_1} \alpha_1 \xrightarrow{\sigma_2} \alpha_2 \ldots,$$

[2] We remark here that while simultaneous execution of the actions also implies that the participants of x finish x synchronously, exit synchronization is not necessary for multiparty interaction; see [17, 32].

where α_0 is an initial state, and each $\alpha_{i-1} \xrightarrow{\sigma_i} \alpha_i$ represents a state transition of the program. Like runs, Π is uniquely determined by the action sequence $\sigma_1\sigma_2\ldots$ executed in Π (assuming some fixed initial state). So we often write Π as $\sigma_1\sigma_2\ldots$. A computation is *complete* if it is infinite or it ends up in a state in which no action is enabled for execution; otherwise, the computation is called *partial*. Since an idle process can autonomously become ready, in the final state of a finite complete computation all primary processes must be ready. In particular, if there is an enabled interaction, then the computation is *deadlocked*. Unless stated otherwise, we shall consider only scheduling programs that produce no deadlocked computation.

Like the dependency relation we assumed for our abstract model, we also assume a dependency relation "$\prec$" respecting Lamport's "happened-before" relation [34] over the set of actions of $\mathbb{SP}_{\mathbb{IS}}$. Two computations Π and Ψ are *equivalent*, denoted by $\Pi \equiv \Psi$, iff they differ up to the order of independent actions.

It should be noted that under the *minimal progress assumption* [41]—any process with an enabled action will eventually execute some action—actions of form (2) do not fully respect Assumption A2. This is because a scheduling program may simply wait until all processes become ready, and then decide on an interaction for execution. To avoid this, we consider only *undelayed* scheduling programs where the establishment of enabled interactions does not depend on idle processes to become ready.

DEFINITION 2.2. A scheduling program $\mathbb{SP}_{\mathbb{IS}}$ is *undelayed* iff for every partial computation Π, if there is an interaction enabled in (the last state of) Π, then Π has a continuation such that some interaction will be executed and no process makes a ready transition in the interim.

To abstract runs from computations, we introduce the following definitions. Let $\Pi = \sigma_1\sigma_2\ldots\sigma_n\ldots$ be a computation of $\mathbb{SP}_{\mathbb{IS}}$. Suppose that σ_n denotes the execution of some instance of x. Then, prior to σ_n all participants of x must have committed to x (i.e., with their $p.commit$ variables set to x). Let σ_j be the first commitment. Then we say that the execution of σ_j *establishes* the instance of x. The run corresponding to Π, denoted by $[\Pi]_{\mathbb{IS}}$, is $[\sigma_1]_{\mathbb{IS}}[\sigma_2]_{\mathbb{IS}}\ldots[\sigma_n]_{\mathbb{IS}}\ldots$, where $[\sigma_i]_{\mathbb{IS}}$ is defined as follows:

1. $[\sigma_i]_{\mathbb{IS}} = p.aim$ if the execution of σ_i results in p's transition into a state ready for the set *aim* of interactions,
2. $[\sigma_i]_{\mathbb{IS}} = x$ if the execution of σ_i establishes an instance of x, and
3. $[\sigma_i]_{\mathbb{IS}} = \epsilon$ otherwise.

We say that computation Π is $\mathbb{C}$-*valid* (or simply *valid* or *fair* when the context is clear) if run $[\Pi]_{\mathbb{IS}}$ is $\mathbb{C}$-valid. The implementability of fairness notions is defined as follows.

DEFINITION 2.3. A fairness notion $\mathbb{C}$ is *implementable* for $\mathbb{IS}$ iff there exists an undelayed scheduling program $\mathbb{SP}_{\mathbb{IS}}$ such that for every complete computation Π of $\mathbb{SP}_{\mathbb{IS}}$, $[\Pi]_{\mathbb{IS}} \in \mathbb{C}(\mathbb{IS})$. $\mathbb{C}$ is *implementable* iff $\mathbb{C}$ is implementable for every $\mathbb{IS}$.

Note that the synchronization and mutual exclusion requirements for multiparty interactions have been assumed by a scheduling program via the use of *commit* variables in the program.

3 THE CRITERION

The fairness implementability criterion depends on a notion of strong feasibility and an indistinguishableness relation between runs. We begin with strong feasibility.

DEFINITION 3.1. A fairness notion $\mathbb{C}$ is *strongly feasible* for $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ iff there exists a nonempty subset $\mathbb{S}$ of $\mathbb{C}(\mathbb{IS})$ such that for every run $\rho \in \mathbb{S}$ and every finite prefix π of ρ, the following two conditions are satisfied:

1. Let s be the last state of π. If p is idle in s, then $\pi \cdot p.\mathsf{M}(p, [s]_{hist,p})$ can be extended to a run in $\mathbb{S}$.
2. If some interaction is enabled in s, then there exists an interaction x such that $\pi \cdot x$ can be extended to a run in $\mathbb{S}$.

Intuitively, condition 1 together with the fact $\mathbb{S} \neq \emptyset$ means that an idle process may become ready at any time it wishes. Condition 2 means that when some interaction is enabled, there should be a continuation allowing some interaction to be executed regardless of whether idle processes will become ready.

Note that we do not require $\mathbb{S} = \mathbb{C}(\mathbb{IS})$. This is because for a scheduling program $\mathbb{SP}_{\mathbb{IS}}$ to implement $\mathbb{C}$, it suffices that every computation of $\mathbb{SP}_{\mathbb{IS}}$ is valid; there is no need for $\mathbb{SP}_{\mathbb{IS}}$ to generate all possible valid computations.[3] For example, if both x and y are enabled at the end of a partial run π, and the fairness notion $\mathbb{C}$ permits either one to be continued, then an implementation of $\mathbb{C}$ can decide to let one of the two interactions, say x, as the only continuation. Moreover, let $\mathbb{C}'$ be a fairness notion such that $\mathbb{C}'(\mathbb{IS})$ contains only runs $[\Pi]_{\mathbb{IS}}$, where Π is a computation of $\mathbb{SP}_{\mathbb{IS}}$. Then $\mathbb{C}'$ can also be implemented by $\mathbb{SP}_{\mathbb{IS}}$. Observe that $\pi \cdot x$ and $\pi \cdot y$ are partial runs of $\mathbb{IS}$ (where π is the above partial computation which ends up with a state where x and y are both enabled) but $\pi \cdot y$ does not have a continuation to a $\mathbb{C}'$-valid run. Therefore, from the implementation's concern we do not need every partial run of $\mathbb{IS}$ be extended to a valid one. This is the main difference between our feasibility and the notion of feasibility proposed by Apt *et al.* [4].

From a more operational standpoint, strong feasibility can be exhibited by an explicit scheduler to schedule the behavior of the processes. Unlike those used in [4, 40], however, the scheduler here should only determine for ready processes which interactions to execute; the transitions from idle to ready are given independently by an adversary to capture the processes' autonomy in making these transitions. Thus, a run is the result of a 2-player game between the explicit scheduler and a given adversary. The following definition is used to realize this.

DEFINITION 3.2. 1. An *adversary* A for $\mathbb{IS}$ is a function which given a run $\pi \in \text{run}^*(\mathbb{IS})$ returns either an empty sequence ϵ or a sequence of actions $p_1.I_1 \ldots p_k.I_k$ as the continuation of π such that $\pi \cdot p_1.I_1 \ldots p_k.I_k$ represents a legal run of $\mathbb{IS}$. Moreover, $A(\pi) = \epsilon$ only if π is complete or some interaction is enabled in π (i.e., enabled in the last state of π).

2. A *nonblocking scheduler*[4] S for $\mathbb{IS}$ is a function which given a run $\pi \in \text{run}^*(\mathbb{IS})$ returns either ϵ or an interaction x enabled in π as the continuation of π. Moreover, $S(\pi) = \epsilon$ only if no interaction is enabled in π.[5]

3. The result of the game up to round i is defined by $r^i(S, A)$, where

$$r^i(S, A) = \begin{cases} \epsilon: & i = 0 \\ r^{i-1}(S, A) \cdot A(r^{i-1}(S, A)): & i = 2n-1, n \in N \\ r^{i-1}(S, A) \cdot S(r^{i-1}(S, A)): & i = 2n, n \in N. \end{cases}$$

The run generated by S versus A, denoted by $r(S, A)$, is the result of the game proceeding in maximal rounds.

Note that $r(S, A)$ must be complete. We say that a nonblocking scheduler S satisfies a fairness notion $\mathbb{C}$ if $r(S, A) \in \mathbb{C}(\mathbb{IS})$ for every adversary A. The following proposition follows directly from the above definition.

PROPOSITION 3.1. *A fairness notion $\mathbb{C}$ is strongly feasible for $\mathbb{IS}$ iff there exists a nonblocking scheduler S such that $r(S, A) \in \mathbb{C}(\mathbb{IS})$ for every adversary A.*

Thus, to show that $\mathbb{C}$ is strongly feasible for $\mathbb{IS}$ we need to construct a nonblocking scheduler S such that $r(S, A) \in \mathbb{C}(\mathbb{IS})$ for every adversary A. Note that by Definition 3.2, a nonblocking scheduler S must always return an interaction if it is given a run π in which some interaction is enabled. Otherwise, S versus A would not be able to generate a complete run if A in response refuses to schedule any more

[3] In the terminology of [4], we do not require $\mathbb{SP}_{\mathbb{IS}}$ to be *faithful* to $\mathbb{C}$.

[4] It was referred to as "nonpreemptive scheduler" in [27].

[5] For simplicity, we allow S to schedule only one interaction at a time even if there is more than one nonconflicting interaction enabled. This does not lose any generality because the game allows the adversary in response to suspend idle processes from becoming ready until all enabled interactions have been disabled.

process to enter a ready state. This is why the scheduler is termed "nonblocking." Similarly, $A(\pi)$ must not return an empty sequence when π is partial and contains no enabled interaction, for otherwise no scheduler S versus A could possibly generate a complete run.

To introduce the indistinguishableness relation, we need an operation of interprocess permutation and an operation of retraction. Let

$$\pi = p_{1,1}.I_{1,1} \ldots p_{1,k_1}.I_{1,k_1}x_1 p_{2,1}.I_{2,1} \ldots p_{2,k_2}.I_{2,k_2}x_2 \ldots,$$

where $x_1, x_2, \ldots$ are interactions executed in π. We say that ρ is obtained from π by an *interprocess permutation* if

$$\rho = q_{1,1}.J_{1,1} \ldots q_{1,k_1}.J_{1,k_1}x_1\ q_{2,1}.J_{2,1} \ldots q_{2,k_2}.J_{2,k_2}x_2 \ldots$$

such that for each $i > 0$, $q_{i,1}.J_{i,1}, \ldots, q_{i,k_i}.J_{i,k_i}$ is a permutation of $p_{i,1}.I_{i,1}, \ldots, p_{i,k_i}.I_{i,k_i}$. Furthermore, ψ is obtained from π by a *retraction* if

$$\begin{aligned}\psi = {} & p_{1,1}.I_{1,1} \ldots p_{1,k_1}.I_{1,k_1}p_{2,1}.I_{2,1} \ldots p_{2,h_2}.I_{2,h_2}x_1 \\ & p_{2,h_2+1}.I_{2,h_2+1} \ldots p_{2,k_2}.I_{2,k_2}p_{3,1}.I_{3,1} \ldots p_{3,h_3}.I_{3,h_3}x_2 \\ & p_{3,h_3+1}.I_{3,h_3+1} \ldots p_{3,k_3}.I_{3,k_3}p_{4,1}.I_{4,1} \ldots p_{4,h_4}.I_{4,h_4}x_3 \ldots\end{aligned}$$

such that for each $i > 1$, $p_{i,1}, \ldots, p_{i,h_i} \notin P_{x_{i-1}}$. That is, ψ is obtained from π by moving, for each $i > 0$, some initial sequence (possibly empty) $p_{i,1}.I_{i,1}, \ldots, p_{i,h_i}.I_{i,h_i}$ of $p_{i,1}.I_{i,1} \ldots p_{i,k_i}.I_{i,k_i}$ forward just before x_{i-1}, and the processes $p_{i,1}, \ldots, p_{i,h_i}$ whose ready transitions are moved must not be involved in x_{i-1}.

DEFINITION 3.3. A run ρ is *indistinguishable* from π, denoted by $\rho \rightsquigarrow \pi$, iff ρ can be obtained from π by an interprocess permutation followed by a retraction. The set of runs that are indistinguishable from π is denoted by *indistinct*(π).

Note that since a run π can be obtained from itself by an interprocess permutation and by a retraction, both operations in the above definition can be considered as optional. As a result, $\pi \in$ *indistinct*(π).

To illustrate, consider the four runs

$$\begin{aligned}\pi_1 &= (p_1p_2x_{12}p_3p_4x_{34})^\omega \\ \pi_2 &= (p_1p_2p_3p_4x_{12}x_{34})^\omega \\ \pi_3 &= (p_3p_4p_1p_2x_{12}x_{34})^\omega \\ \pi_4 &= (p_2p_1p_4p_3x_{12}x_{34})^\omega,\end{aligned}$$

where $P_{x_{12}} = \{p_1, p_2\}$ and $P_{x_{34}} = \{p_3, p_4\}$. In this example, $\pi_2 \rightsquigarrow \pi_1$ because π_2 can be obtained from π_1 by moving each occurrence of p_3p_4 ahead of x_{12}. Also, $\pi_3 \rightsquigarrow \pi_2$ because π_3 differs from π_2 only in the permutation of $p_1p_2p_3p_4$. Note that $\pi_1 \not\rightsquigarrow \pi_2$ and $\pi_3 \not\rightsquigarrow \pi_1$. So the indistinguishableness relation is neither symmetric nor transitive. Moreover, $\pi_4 \rightsquigarrow \pi_1$ because π_1 can be transformed into $(p_2p_1x_{12}p_4p_3x_{34})^\omega$ by an interprocess permutation, which in turn can be transformed into π_4 by a retraction.

Observe that if $\rho \rightsquigarrow \pi$ then the two runs must be equivalent. The converse may not necessarily hold, however. This can be illustrated by the above example where π_1 and π_3 are equivalent but they are not indistinguishable from each other. Thus, indistinguishableness is strictly stronger than the equivalence relation defined earlier by permuting independent actions.

The fairness implementability criterion is defined as follows.

DEFINITION 3.4. A fairness notion $\mathbb{C}$ on $\mathbb{IS}$ satisfies the *fairness implementability criterion* iff there exists a nonblocking scheduler S such that *indistinct*$(r(S, A)) \subseteq \mathbb{C}(\mathbb{IS})$ for every adversary A of $\mathbb{IS}$.

We first provide some intuition behind the criterion. Clearly, any implementation of a fairness notion $\mathbb{C}$ for a system $\mathbb{IS}$ must implicitly assume some scheduling policy to decide which action to execute so as to meet the requirement of the fairness notion. Given Assumption A2 that a process can autonomously make a transition into a ready state, the scheduling policy has no control at all on when a process will make a ready transition. Furthermore, when some interaction has been enabled, the scheduling policy must decide on an interaction for execution, regardless of whether idle processes will become ready or not. This is exactly what is captured by the notion of strong feasibility: the nonblocking scheduler abstracts the scheduling policy, while the adversary stands for the processes to decide when they will make a ready transition. So strong feasibility is a necessary condition for the implementability criterion.

Moreover, any coordinator process that is to implement this scheduling policy must first obtain its knowledge of the global state. Recall Assumption A1 that a process's state cannot be instantly observed by another process. So by A1 and A2, it is clear that when the coordinator has locally observed a sequence of actions $p_1.I_1\,p_2.I_2 \ldots p_k.I_k$, the coordinator may not be able to tell the real execution sequence of them. Hence if the coordinator decides to schedule an interaction based on this observation, then the fairness notion should be general enough to consider all other runs that differ from the coordinator's observation only in the ordering of these ready actions as valid. Otherwise, the coordinator cannot correctly implement the fairness notion. Furthermore, when a coordinator learns that some process is idle, this knowledge must be based on some (direct or indirect) communication between the coordinator and the process. By A1 communication takes nonnegligible time. So the information about the process's idleness may be obsolete when it arrives at the coordinator because, by A2, the process can make a ready transition while the information is being delivered. As a result, when the coordinator decides on an interaction, the fairness notion should also allow this decision even if the ready transition actually occurs before the coordinator makes its decision. The type of runs for which a coordinator cannot distinguish from its observation are formally captured by the indistinguishableness relation amongst runs.

The limitations of the coordinators mean that although a coordinator's observation causes it to generate a run π, the coordinator cannot tell whether π or ρ is the run that actually occurs for any $\rho \in \mathit{indistinct}(\pi)$. Clearly, π is the result of the underlying scheduling policy (i.e., a nonblocking scheduler S "simulated" by the coordinator) versus a specific behavior of the processes (i.e., a specific adversary A). As the scheduling policy must also work for all other possible behaviors of the processes, we therefore have that for every possible adversary A of $\mathbb{IS}$, all runs in $\mathit{indistinct}(r(S, A))$ must satisfy $\mathbb{C}$. This is how the fairness implementability criterion is obtained.

We now formally prove that the criterion is sufficient and necessary to guarantee $\mathbb{C}$'s implementability.

THEOREM 3.2. *A fairness notion $\mathbb{C}$ on $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ is implementable iff there exists a nonblocking scheduler S such that $\mathit{indistinct}(r(S, A)) \subseteq \mathbb{C}(\mathbb{IS})$ for every adversary A of $\mathbb{IS}$.*

Proof of the only-if direction. Suppose $\mathbb{SP}_{\mathbb{IS}}$ is an undelayed scheduling program satisfying $\mathbb{C}$. We present a nonblocking scheduler S by defining, given any adversary A, the partial run $r^i(S, A)$ generated by S versus A for each $i \geq 0$. Concomitantly, we construct a partial computation $c^i(S, A)$ of $\mathbb{SP}_{\mathbb{IS}}$ satisfying the following requirements:

R1. $r^i(S, A) = [c^i(S, A)]_{\mathbb{IS}}$.

R2. For all $\psi \in \mathit{indistinct}(r^i(S, A))$ there exists a partial computation Ψ, $\Psi \equiv c^i(S, A)$, such that $\psi = [\Psi]_{\mathbb{IS}}$.

R1 ensures that for every complete run π generated by S there is a computation Π of $\mathbb{SP}_{\mathbb{IS}}$ such that $[\Pi]_{\mathbb{IS}} = \pi$. Since $\mathbb{SP}_{\mathbb{IS}}$ satisfies $\mathbb{C}$, $\pi \in \mathbb{C}(\mathbb{IS})$. Moreover, R2 ensures that $\mathit{indistinct}(\pi) \subseteq \mathbb{C}(\mathbb{IS})$. Thus, the only-if direction of the theorem is established.

The construction of $r^i(S, A)$ and $c^i(S, A)$ is as follows:

Case 0. $i = 0$: $r^0(S, A) = c^0(S, A) = \epsilon$.

Case 1. $i = 2n - 1, n > 0$:

Case 1.1. $A(r^{i-1}(S, A)) = \epsilon$:

$$r^i(S, A) = r^{i-1}(S, A), \quad \text{and} \quad c^i(S, A) = c^{i-1}(S, A).$$

Case 1.2. $A(r^{i-1}(S, A)) = p_1.I_1 \ldots p_k.I_k, k \geq 1$:

$$r^i(S, A) = r^{i-1}(S, A) \cdot p_1.I_1 \ldots p_k.I_k, \quad \text{and} \quad c^i(S, A) = c^{i-1}(S, A) \cdot p_1.I_1 \ldots p_k.I_k.$$

Note that for notational simplicity, here we also use $p_i.I_i$ to denote the action of $\mathbb{SP}_{\mathbb{IS}}$ corresponding to process p_i's transition into a state ready for the set I_i of interactions.

Case 2. $i = 2n, n > 0$:

Case 2.1. No interaction is enabled in $r^{i-1}(S, A)$:

$$r^i(S, A) = r^{i-1}(S, A), \quad \text{and} \quad c^i(S, A) = c^{i-1}(S, A).$$

Case 2.2. Some interaction is enabled in $r^{i-1}(S, A)$: Let $\sigma_1 \ldots \sigma_j$ be a continuation of $c^{i-1}(S, A)$ such that the execution of σ_j causes some process to commit to an interaction, say x, and in the interim no primary process makes a ready transition. Since some process has committed to x, by the assumption on $\mathbb{SP}_{\mathbb{IS}}$, $c^{i-1}(S, A) \cdot \sigma_1 \ldots \sigma_j$ can be extended further to $c^{i-1}(S, A) \cdot \sigma_1 \ldots \sigma_j \sigma_{j+1} \ldots \sigma_{j+k}$ such that all other participants of x will also commit to x, and no primary process other than the participants of x is involved in the computation $\sigma_{j+1} \ldots \sigma_{j+k}$. Moreover, since after σ_{j+k} all participants of x have committed to x, an instance of x can be executed. Let σ_{j+k+1} denote the action of this execution. Then,

$$r^i(S, A) = r^{i-1}(S, A) \cdot x \quad \text{and} \quad c^i(S, A) = c^{i-1}(S, A) \cdot \sigma_1 \ldots \sigma_{j+k} \sigma_{j+k+1}.$$

To complete the proof of the only-if direction we shall show that for all $i \geq 0$ the following conditions hold:

(i) $r^i(S, A)$ represents a legal run.

(ii) If some interaction is enabled in $r^{i-1}(S, A)$, where $i = 2n$ for some $n > 0$, then $r^i(S, A) = r^{i-1}(S, A) \cdot x$ for some $x \in \mathsf{I}$.

(iii) $c^i(S, A)$ represents a legal computation of $\mathbb{SP}_{\mathbb{IS}}$.

(iv) let x be the last interaction executed in $c^i(S, A)$, and let σ_l denote the execution of x. Let σ_j be the action that establishes this instance of x. Then no primary process other than the participants of x is involved in the computation from σ_j to σ_l.

(v) R1 holds; that is, $r^i(S, A) = [c^i(S, A)]_{\mathbb{IS}}$.

(vi) R2 holds; that is, for all $\psi \in \mathit{indistinct}(r^i(S, A))$, there exists a partial computation Ψ, $\Psi \equiv c^i(S, A)$, such that $\psi = [\Psi]_{\mathbb{IS}}$.

The first two conditions ensure that S is nonblocking; together with the third condition they ensure that the construction results in legal runs and legal computations. The last two conditions guarantee R1 and R2. Condition (iv) is used to help assert condition (vi).

We prove the above six conditions by induction on i. It is easy to see that the six conditions hold for $i = 0$. Assume the induction hypothesis that they hold up to $i = m - 1$, $m \geq 1$. To show that they hold

for $i = m$, we can divide the problem into four cases based on the construction of $r^i(S, A)$ and $c^i(S, A)$:

(1) $m = 2n - 1, n > 0$:
  (1.1) $A(r^{m-1}(S, A)) = \epsilon$,
  (1.2) $A(r^{m-1}(S, A)) = p_1.I_1 \ldots p_k.I_k, k \geq 1$.
(2) $m = 2n, n > 0$:
  (2.1) no interaction is enabled in $r^{m-1}(S, A)$
  (2.2) some interaction is enabled in $r^{m-1}(S, A)$

For case (1.1), since by the construction we have $r^m(S, A) = r^{m-1}(S, A)$ and $c^m(S, A) = c^{m-1}(S, A)$, the induction hypothesis, together with the fact that condition (ii) holds vacuously because $m = 2n-1$, implies that the six conditions hold for this case as well.

For case (1.2), by the construction we have $r^m(S, A) = r^{m-1}(S, A) \cdot p_1.I_1 \ldots p_k.I_k$, and $c^m(S, A) = c^{m-1}(S, A) \cdot p_1.I_1 \ldots p_k.I_k$. Since A is an adversary for $\mathbb{IS}$, by definition, $r^m(S, A)$ must be a legal run of $\mathbb{IS}$ (given the induction hypothesis that $r^{m-1}(S, A)$ is a legal run of $\mathbb{IS}$). In particular, $p_1, \ldots, p_k$ must be idle in $c^{m-1}(S, A)$, and so they are eligible to make their ready transitions after the partial computation $c^{m-1}(S, A)$. So conditions (i) and (iii) are satisfied. Condition (ii) is satisfied vacuously because $m = 2n - 1$. Since only ready transitions are added to $r^{m-1}(S, A)$, condition (iv) follows directly from the induction hypothesis. It is also easy to see that the construction guarantees condition (v). So it remains to show that condition (vi) is satisfied for this case.

Let ψ be a run in $\mathit{indistinct}(r^m(S, A))$. By definition, ψ is obtained from $r^{m-1}(S, A) \cdot p_1.I_1 \ldots p_k.I_k$ by an interprocess permutation followed by a retraction. There are three cases to consider:

- The operations of interprocess permutation and retraction do not involve any of the new ready transitions $p_1.I_1, \ldots, p_k.I_k$.
- Only the operation of interprocess permutation involves the ready transitions $p_1.I_1, \ldots, p_k.I_k$.
- Otherwise; i.e., the operation of retraction involves the ready transitions $p_1.I_1, \ldots, p_k.I_k$.

We shall consider here only the last case; the other two cases can be treated analogously (but simpler). In the last case $r^{m-1}(S, A)$ must contain some interaction, for otherwise the operation of retraction would be meaningless. Let x be the last interaction executed in $r^{m-1}(S, A)$. Then either (a) $r^{m-1}(S, A) = \psi' \cdot x$ or (b) $r^{m-1}(S, A) = \psi' \cdot x \cdot p'_1.I'_1 \ldots p'_g.I'_g$. Again, we shall consider only the latter case; the former case can be treated analogously.

We first note that the set of processes $\{p'_1, \ldots, p'_g\}$ must be disjoint with $\{p_1, \ldots, p_k\}$, and $\psi' \cdot x = r^l(S, A)$ for some $l < m$. Since $\psi \in \mathit{indistinct}(r^m(S, A))$, ψ can be written as $\psi'' \cdot q_1.J_1 \ldots q_h.J_h \cdot x \cdot q_{h+1}.J_{h+1} \ldots q_{g+k}.J_{g+k}$ such that $q_1.J_1, \ldots, q_{g+k}.J_{g+k}$ is a permutation of $p'_1.I'_1, \ldots, p'_g.I'_g, p_1.I_1, \ldots, p_k.I_k$, and x does not involve $q_1, \ldots, q_h$. Clearly, $\psi'' \cdot x \in \mathit{indistinct}(\psi' \cdot x)$. Given that $\psi' \cdot x = r^l(S, A)$, by conditions (iv) and (v) of the induction hypothesis, there exists some partial computation Γ of $\mathbb{SP}_{\mathbb{IS}}$ such that $c^l(S, A) = \Gamma \cdot \sigma_1 \ldots \sigma_f \sigma_{f+1}$, where σ_1 establishes an instance of x, the execution of subsequent actions $\sigma_2, \ldots, \sigma_f$ causes every participant of x to commit to x, and σ_{f+1} corresponds to the execution of x. Note that no primary process other than that of P_x is involved in these actions $\sigma_1, \ldots, \sigma_{f+1}$. Also, by the construction, $c^m(S, A) = \Gamma \cdot \sigma_1 \ldots \sigma_f \sigma_{f+1} p'_1.I'_1 \ldots p'_g.I'_g, p_1.I_1 \ldots, p_k.I_k$. By condition (vi) of the induction hypothesis, there exists Ψ' such that $\Psi' \equiv c^l(S, A)$ and $\psi'' \cdot x = [\Psi']_{\mathbb{IS}}$. Given that $c^l(S, A) = \Gamma \cdot \sigma_1 \ldots \sigma_f \sigma_{f+1}$, there exists some Γ' such that $\Gamma' \equiv \Gamma$, $\Psi' = \Gamma' \cdot \sigma_1 \ldots \sigma_f \sigma_{f+1}$, and $\psi'' \cdot x = [\Gamma' \cdot \sigma_1 \ldots \sigma_f \sigma_{f+1}]_{\mathbb{IS}}$. So $\Gamma' \cdot \sigma_1 \ldots \sigma_f \sigma_{f+1} \cdot q_1.J_1 \ldots\ldots q_h.J_h \cdot q_{h+1}.J_{h+1} \ldots q_{g+k}.J_{g+k}$ must also be a legal computation and is equivalent to $c^m(S, A)$. Moreover, since the actions $\sigma_1, \ldots, \sigma_f, \sigma_{f+1}$ involve only primary processes in P_x, and since x does not involve $q_1, \ldots, q_h$, the computation $\Psi = \Gamma' \cdot q_1.J_1 \ldots q_h.J_h \cdot \sigma_1 \ldots \sigma_f \sigma_{f+1} \cdot q_{h+1}.J_{h+1} \ldots q_{g+k}.J_{g+k}$ is still legal and equivalent to $c^m(S, A)$. It is easy to see that $\psi = \psi'' \cdot q_1.J_1 \ldots q_h.J_h \cdot x \cdot q_{h+1}.J_{h+1} \ldots q_{g+k}.J_{g+k} = [\Psi]_{\mathbb{IS}}$. So condition (vi) is satisfied. This completes the proof of case 1.2.

For case (2.1), we have $r^m(S, A) = r^{m-1}(S, A)$ and $c^m(S, A) = c^{m-1}(S, A)$. The induction hypothesis together with the fact that condition (ii) holds vacuously because no interaction is enabled in $r^{m-1}(S, A)$ implies that the six conditions hold for this case.

For case (2.2), since by condition (v) of the induction hypothesis $r^{m-1}(S, A) = [c^{m-1}(S, A)]_{\mathbb{IS}}$, if some interaction is enabled in $r^{m-1}(S, A)$, then it must also be enabled in $c^{m-1}(S, A)$. Moreover, since

$\mathbb{SP}_{\mathbb{IS}}$ is undelayed, if some interaction is enabled in $c^{m-1}(S, A)$, then $c^{m-1}(S, A)$ has a continuation $\sigma_1 \ldots \sigma_j$ (possibly more than one) such that the execution of σ_j causes some process to commit to an interaction. So the construction of $c^m(S, A)$ from $c^{m-1}(S, A)$ guarantees that $c^m(S, A)$ is a legal computation of $\mathbb{SP}_{\mathbb{IS}}$. Likewise, $r^m(S, A)$ is also a legal run of $\mathbb{IS}$. So conditions (i), (ii), and (iii) are satisfied for this case. It is also easy to see that the construction of $r^m(S, A)$ and $c^m(S, A)$ guarantees conditions (iv) and (v). For condition (vi), observe that if $\psi \in \mathit{indistinct}(r^m(S, A))$, then the operations of interprocess permutation and retraction to transform $r^m(S, A) = r^{m-1}(S, A) \cdot x$ to ψ must not involve x. So ψ can be written as $\psi' \cdot x$ such that $\psi' \in \mathit{indistinct}(r^{m-1}(S, A))$. The induction hypothesis then implies that condition (vi) holds for this case as well.

This completes the proof of the only-if direction. ■

Proof of the if-direction. Suppose there exists a nonblocking scheduler S such that for every adversary A, $\mathit{indistinct}(r(S, A)) \subseteq \mathbb{C}(\mathbb{IS})$. We present an undelayed scheduling program $\mathit{Simulate}(S)$ which employs a coordinator to simulate the behavior of S (see Fig. 3). Like S, the coordinator proceeds in rounds. In each round, it first waits for idle processes to inform it of their readiness. Each process p is required to send a message $\mathit{Ready}(p, I)$ to the coordinator when it makes a ready transition $p.I$.

When the coordinator learns that some interaction has been enabled, it initiates a *querying procedure*, attempting to confirm if the other processes which have not yet informed the coordinator of their readiness are indeed idle. To do so, the coordinator sends a query message to each of them and waits for the response. The querying procedure terminates if every queried process replies a message *idle* to the query indicating that the process was idle when it received the query. If some process responds with $\mathit{Ready}(p, I)$, then the coordinator has to reinitiate a querying procedure. Note that if a new querying procedure is necessary, then the number of processes that are idle (to the coordinator's knowledge) must be decreased by at least one. Since the total number of processes in the system is finite, eventually no more querying procedure will be needed.

When the coordinator has finished its querying procedures, it determines an interaction for execution by simulating the scheduling of S. Let $\mathit{Ready}(p_{i,1}, I_{i,1}), \ldots, \mathit{Ready}(p_{i,k}, I_{i,k_i})$ be the sequence of ready messages the coordinator receives in this round. Then, in the simulation the coordinator assumes that the adversary A provides the sequence of ready transitions $p_{i,1}.I_{i,1} \ldots p_{i,k_i}.I_{i,k_i}$ to S. Let x_i be the interaction chosen by S. Then the coordinator finishes this round by sending a message $\mathit{Commit}(x_i)$ to inform each process in P_{x_i} to execute x_i. The commit messages are acknowledged by the receivers. Note that if some interaction is enabled, then S (and thus the coordinator) must schedule an interaction for execution because S is nonblocking. So $\mathit{Simulate}(S)$ is undelayed.

It is easy to see that the algorithm presented in Fig. 3 complies with the restrictions of the implementation model described in Section 2.2.

To show that $\mathbb{C}$ is implemented by $\mathit{Simulate}(S)$, we need to show $[\Pi]_{\mathbb{IS}} \in \mathbb{C}(\mathbb{IS})$ for every computation Π of $\mathit{Simulate}(S)$. Recall that for each Π, the coordinator of $\mathit{Simulate}(S)$ has assumed an adversary A with which S is playing. Let $r(S, A)$ be the complete run generated by S versus A. We shall show that $[\Pi]_{\mathbb{IS}} \in \mathit{indistinct}(r(S, A))$. Since $\mathit{indistinct}(r(S, A)) \subseteq \mathbb{C}(\mathbb{IS})$, we have that $[\Pi]_{\mathbb{IS}} \in \mathbb{C}(\mathbb{IS})$.

We begin by defining a mapping σ from actions in $r(S, A)$ to actions in Π. Recall that the coordinator appends a ready transition $p.I$ to some prefix ρ of $r(S, A)$ if, and only if, it has received a message $\mathit{Ready}(p, I)$ from p. Process p sends this message because it has made a ready transition in $\mathbb{SP}_{\mathbb{IS}}$. We use $\sigma(p.I)$ to denote this ready transition. Moreover, the coordinator appends an interaction x to some prefix ρ of $r(S, A)$ because $S(\rho) = x$. To actually schedule x, the coordinator sends a message $\mathit{Commit}(x)$ to each participant of x. Each participant, upon receiving the commit message, will set its variable commit to x. Let $\sigma(x)$ denote the first reception of the commit messages. By definition of $[\Pi]_{\mathbb{IS}}$, only the actions in Π which can be mapped from the actions in $r(S, A)$ are relevant to the projection from Π to $[\Pi]_{\mathbb{IS}}$. Note that the mapping σ preserves the dependency relation of the actions in $r(S, A)$ in the sense that if $a \prec b$ then $\sigma(a) \prec \sigma(b)$ in Π.

We claim that $\mathit{Simulate}(S)$ guarantees the following conditions:

C1. The sequence of interactions executed in $[\Pi]_{\mathbb{IS}}$ is the same as the sequence of interactions executed in $r(S, A)$.

C2. Let a be a ready transition in $r(S, A)$, and assume that a occurs between two interactions x_1 and x_2 in $r(S, A)$. Furthermore, let b be the action just before x_1. Then the four actions $\sigma(b), \sigma(a), \sigma(x_1)$,

The code of each process p :

{ $state = idle \longrightarrow$ /* make a ready transition $p.aim$ */
  $state := ready$; $aim := \mathsf{M}(p, hist)$;
  send a message $Ready(p, aim)$ to the coordinator
□ receive a query message from the coordinator $\longrightarrow$
  if $state = idle$, reply to the query with a message $idle$;
  otherwise, ignore this query (because the process has already sent a ready message to the coordinator when it entered its ready state.)
□ receive $Commit(x)$ from the coordinator $\longrightarrow$
  $commit := x$; acknowledge the message;
□ $commit = x \longrightarrow$ /* participate in x */
  execute x; $state := idle$; $aim := \emptyset$; $hist := hist \cdot x$; $commit := \perp$;
}

The code of the coordinator :

variables :
new_ready: a queue of elements $p.I$ indicating that the coordinator has received message $Ready(p, I)$ in current round;
yet_handled: a set of elements $p.I$ indicating that the coordinator has received message $Ready(p, I)$ but has not yet scheduled any interaction for process p;
active: a flag indicating if the coordinator has learned that an interaction is enabled;
i: the number of rounds that have proceeded so far;

{ not *active*, receive message $Ready(p, I) \longrightarrow$
  add $p.I$ to *new_ready* and *yet_handled*;
  if some interaction has been enabled, then set *active* to true.
□ *active* $\longrightarrow$
  while *true* do {
    let $Q = \mathsf{P} - \{p \mid p.I \in yet\_handled\}$;
    send query messages to each process in Q and wait for the responses;
    if $Q = \emptyset$ or all the queries are answered *idle*, then exit this while-loop;
      otherwise, for each response that is of type $Ready(p, I)$
      add $p.I$ to *new_ready* and *yet_handled* }
  in simulating the game of S versus A, assume that the adversary
    assigns *new_ready* as the ready transitions in round i;
  let x_i be the interaction chosen by S; then for each $p \in P_{x_i}$ do {
    send $Commit(x_i)$ to p;
    delete the entry $p.I$ from *yet_handled*; }
  wait for acknowledgments to the commit messages;
  new_ready $:= \emptyset$; $i := i + 1$;
  if no other interaction is enabled, then *active* := false;
}

FIG. 3. The scheduling program *Simulate*(S).

and $\sigma(x_2)$ have the dependency relations

$$\sigma(b) \prec \sigma(a) \qquad \text{and} \qquad \sigma(a) \prec \sigma(x_2).$$

Note that if there is no x_1 (because a belongs to the initial ready transitions of $r(S, A)$), then, of course, only the relation $\sigma(a) \prec \sigma(x_2)$ will be considered. Similarly, if there is no x_2 (because $r(S, A)$ is a finite complete run), then only $\sigma(b) \prec \sigma(a)$ will be considered.

It can be seen that C1, C2, and the fact that actions in Π respect the dependency relation in $r(S, A)$ ensure that $[\Pi]_{\mathbb{IS}} \in indistinct(r(S, A))$.

To see C1, let x_1 and x_2 be any two interactions in $r(S, A)$, and assume that x_1 occurs before x_2. Consider the actions in Π. By the scheduling program, when the coordinator decides to schedule x_1, it will send commit messages to every participant of x_1 and wait for the acknowledgments. So every participant of x_1 must have received the commit message before the coordinator wishes to schedule x_2. So the action $\sigma(x_1)$ must occur causally before $\sigma(x_2)$.[6]

To see C2, let a be a ready transition in $r(S, A)$ and assume that $a = p.I$. Consider first that some interaction occurs before a, and let x_1 be the latest interaction. Let b be the action just before x_1. Observe that the coordinator adds a to $r(S, A)$ because p has made a ready transition and has sent a message *Ready*(p, I) to the coordinator. (In the algorithm each ready transition is associated with a unique ready message.) Since x_1 occurs before a, the ready message is received after the coordinator decides to schedule x_1. That is, the coordinator considers p as idle just before it decides to schedule x_1. However, before the coordinator decides to schedule x_1, it must have completed a querying procedure to make sure that every process to which it considers as idle has replied an idle message to its query. So between the two actions $\sigma(b)$ and $\sigma(x_1)$ in Π, the coordinator has sent a query message to p and p has replied idle to the query. This means that action $\sigma(a)$ whose execution resulting in the ready transition a must occur causally after $\sigma(b)$; that is, $\sigma(b) \prec \sigma(a)$.

Consider next that some interaction occurs after a, and let x_2 be the first such interaction. Then, it is clear that action $\sigma(a)$ must occur causally before the coordinator receives the corresponding message *Ready*(p, I), and the reception must occur causally before the coordinator issues a commit message to any member of x_2. So $\sigma(a) \prec \sigma(x_2)$.

This completes the if-direction of the proof. ■

We note that the above proof does not rely on how long it takes to deliver a message. Thus, the theorem holds as well if the transmission delay is finitely bounded.

4 PROPERTIES OF IMPLEMENTABLE FAIRNESS NOTIONS

In this section we provide some useful lemmas derived from the fairness implementability criterion. Recall from Section 3 that if $\rho \rightsquigarrow \pi$, then ρ and π must be equivalent. If equivalent runs are either all valid or all invalid, then *indistinct*(π) contains either all valid runs or all invalid runs. Therefore, if the fairness notion in consideration is also strongly feasible, then by Theorem 3.2 it must be implementable. We thus have the following lemma.

LEMMA 4.1. *If $\mathbb{C}$ is strongly feasible and equivalence-robust for $\mathbb{IS}$, then $\mathbb{C}$ is implementable for $\mathbb{IS}$.*

Clearly, the above lemma does not rule out the possibility of a non-equivalence-robust (but strongly feasible) fairness notion being implementable. Similarly, a fairness notion $\mathbb{C}$ may still be implementable even if $\mathbb{C}(\mathbb{IS})$ contains some π such that *indistinct*$(\pi) \not\subseteq \mathbb{C}(\mathbb{IS})$. The crux is to find a nonblocking scheduler S that can avoid generating "odd" runs like π whose *indistinct*(π) contains an invalid run. (Notice the existential quantifier in Theorem 3.2.) Section 5.3 presents an example for this.

However, there are runs that cannot be avoided by any nonblocking scheduler. So if these runs happen to be "odd," then the fairness notion in question is not possible. *Singular* runs, as defined below, are an example of runs that must be generated by every nonblocking scheduler.

DEFINITION 4.1. A run π is *singular* iff in every state of the run at most one interaction is enabled.

If a nonblocking scheduler faces a situation in which only one interaction is enabled, then by definition the scheduler must select it for execution. So every nonblocking scheduler for $\mathbb{IS}$ must generate all singular runs of $\mathbb{IS}$.

LEMMA 4.2. *If $\pi \in$ run$(\mathbb{IS})$ is singular, then for every nonblocking scheduler S of $\mathbb{IS}$, there exists an adversary A such that $\pi = r(S, A)$.*

Proof. The singular run π itself expresses the behavior of the adversary: Let π be

$$a_{1,1} \ldots a_{1,n_1} x_1 a_{2,1} \ldots a_{2,n_2} x_2 \ldots,$$

[6] It can be seen that the order of interactions in $r(S, A)$ can still be preserved in Π even if only one participant of every scheduled interaction needs to acknowledge the coordinator's commit messages.

where each $a_{i,1} \ldots a_{i,n_i}$ denotes a sequence of ready transitions and x_i denotes interaction execution. In round i the adversary schedules the sequence $a_{i,1} \ldots a_{i,n_i}$. Since only x_i is enabled at this point, the scheduler in turn must schedule x_i for execution. Hence, S versus A generates exactly the run. ■

Therefore, if $\mathbb{C}$ treats some run indistinguishable from a singular run as invalid, then by Theorem 3.2, $\mathbb{C}$ must not be implementable. This is stated in the following lemma and, as we shall see in the following section, is very useful in proving the unimplementability of fairness notions.

LEMMA 4.3. *If there exists a singular run* $\pi \in run(\mathbb{IS})$ *such that indistinct*$(\pi) \not\subseteq \mathbb{C}(\mathbb{IS})$, *then* $\mathbb{C}$ *is not implementable for* $\mathbb{IS}$.

Proof. By Lemma 4.2, every nonblocking scheduler for $\mathbb{IS}$ must generate π. However, since *indistinct*$(\pi) \not\subseteq \mathbb{C}(\mathbb{IS})$, by Theorem 3.2, therefore, no undelayed scheduling program for $\mathbb{IS}$ can satisfy $\mathbb{C}$. ■

5 APPLICATIONS OF THE CRITERION

In this section we use the proposed criterion to examine several fairness notions that are typically associated with multiparty interactions. In particular, if a fairness notion is not implementable, we wish to identify the system structure that renders the impossibility phenomenon. For this, we shall consider interaction systems whose programs are of type $\mathsf{M}^{\forall}$. The fairness notions to be examined include *strong interaction fairness* (SIF), *strong process fairness* (SPF), *weak process fairness* (WPF), *U-fairness* [5], and *hyperfairness* [6].

51. Strong Interaction Fairness

Recall that SIF requires an interaction that is enabled infinitely often to be executed infinitely often. Using Lemma 4.3, we can establish an impossibility result for SIF. For intuition, define *permute*(π) to be the set of runs that can be obtained from π by an interprocess permutation. Clearly, if π satisfies SIF, then all runs in *permute*(π) satisfy SIF. Therefore, if π satisfies SIF but some run indistinguishable from π does not, then there must exist some $\pi' \in$ *permute*(π) such that π' contains infinitely many sequences of the form

$$yq_1q_2 \ldots q_k,$$

where $k \geq 1$, $q_1, q_2, \ldots q_k \notin P_y$, such that moving $q_1q_2 \ldots q_k$ forward ahead of y (i.e., deferring y until q_k) causes some interaction x, which has been enabled only a finite number of times in π', to be enabled immediately before y is executed. So $P_x \cap P_y \neq \emptyset$ and $\{q_1, q_2, \ldots, q_k\} \cap P_x \neq \emptyset$. Given that $q_1, q_2, \ldots q_k \notin P_y$, we have $P_x - P_y \neq \emptyset$.

When two sequences of the form $yq_1q_2 \ldots q_k \ldots$ are placed next to each other, the resulting sequence $yq_1q_2 \ldots q_k \ldots yq_1q_2 \ldots q_k \ldots$ now contains a subsequence $q_1q_2 \ldots q_k \ldots y$. Hence x is immediately enabled before y is executed, unless a third interaction z is placed in between $q_1q_2 \ldots q_k$ and y. Given that x is enabled only a finite number of times, such an interaction exists, and $(P_x - P_y) \cap P_z \neq \emptyset$; that is, $P_x \cap P_z \not\subseteq P_y$ (which subsumes the previous condition that $P_x - P_y \neq \emptyset$). Let z be the first such interaction that is executed after the sequence $yq_1q_2 \ldots q_k$.

Taking z into account, π' contains infinitely many sequences of the form $yq_1q_2 \ldots q_k \ldots z \ldots$. Since prior to the execution of z all processes in $P_x - P_y$ are ready, $P_x \cap P_y$ cannot be all ready before z is executed. Otherwise, x would be enabled immediately before z is executed, and so would be enabled infinitely often throughout π'. So $P_x \cap P_y \not\subseteq P_z$ (which subsumes the previous condition that $P_x \cap P_y \neq \emptyset$).[7]

Hence, if the interaction structure of the underlying system can satisfy the above conditions and a run like π is inevitable (e.g., π is singular) to every nonblocking scheduler, then SIF will be impossible for the system. Indeed, such a setting is possible, as can be illustrated by the system $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^{\forall})$, where I has the structure as depicted in Fig. 4c. Let $\pi = (p_1p_3yp_2p_4z)^{\omega}$ and $\rho = (p_1p_3p_2yp_4z)^{\omega}$. Then, π

[7] It should be noted that the condition $P_x \cap P_y \not\subseteq P_z$ is obtained only because the definition of $\mathsf{M}^{\forall}$ lets all processes in $P_x \cap P_y$ be ready to execute x whenever they are ready.

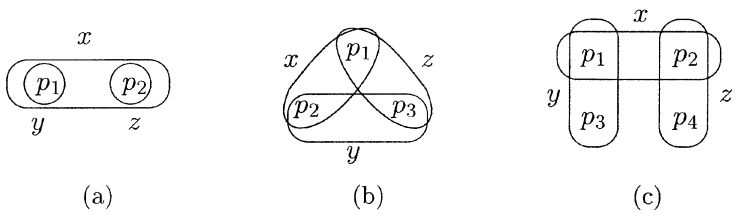

FIG. 4. Interaction structures for which SIF is not possible.

satisfies SIF and is singular, and $\rho \rightsquigarrow \pi$. However, ρ does not satisfy SIF because x is enabled each time p_2 is ready but x is never executed. So by Lemma 4.3 SIF is not possible for the system. The following theorem characterizes the interaction structure for which SIF is not possible.

THEOREM 5.1. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^{\forall})$. *Assume* $\exists\, x, y, z, \in \mathsf{I}$, $P_x \cap P_y \not\subseteq P_z$ *and* $P_x \cap P_z \not\subseteq P_y$. *Then SIF cannot be implemented for* $\mathbb{IS}$.

Proof. Let S be a nonblocking scheduler S for $\mathbb{IS}$ that satisfies SIF. We present an adversary A such that some run in *indistinct*$(r(S, A))$ does not satisfy SIF. By Theorem 3.2, therefore, there is no scheduling program for $\mathbb{IS}$ satisfying SIF. A behaves as follows.

1. Initially, A schedules all processes in $(P_x - P_z) \cup P_y$ to enter a ready state. (The order of the state transitions is rather arbitrary.) By the definition of $\mathsf{M}^{\forall}$, y is enabled (and possibly some others, too) but x is not due to the lack of some process in $P_x \cap P_z - P_y$. (Note that $P_x \cap P_z - P_y \neq \emptyset$ because $P_x \cap P_z \not\subseteq P_y$.)

2. A's subsequent behavior then depends on S's reaction.

2.1. If S selects y for execution, then A in turn schedules the processes in P_z to enter a ready state. So z becomes enabled, but still x is disabled due to the lack of some process in $P_x \cap P_y - P_z$.

2.1.1. If S next selects z for execution, then A schedules the processes in P_y to become ready and waits for S's response. Note that at this point y is enabled again, but still x is disabled due to the lack of some process in $P_x \cap P_z - P_y$. The following behavior of A is the same as the beginning of Step 2.

2.1.2. If, however, S selects some interaction v instead of z, then A in response schedules the processes in P_v to be ready again so that z is enabled in S's next turn. If subsequently z is chosen by S, then A schedules the processes in P_y to enter a ready state as described in 2.1.1. Otherwise, A continues to schedule the set of processes for which S has just selected for interaction to enter a ready state. Note that in this tournament z will eventually be chosen because S satisfies SIF. Moreover, x remains disabled due to the lack of some process in $P_x \cap P_y - P_z$.

2.2. If S selects some other interaction u instead of y, then A in response schedules the processes in P_u to become ready so that y is enabled again in S's next turn. If y is finally chosen by S, then A behaves as that described in step 2.1. Otherwise, A continues to schedule the set of processes for which S has just selected for interaction to become ready. As discussed above, in this tournament y will eventually be selected because S satisfies SIF. Also, x remains disabled due to the lack of some process in $P_x \cap P_z - P_y$.

Therefore, S versus A must generate a run in which y is executed infinitely often because y is enabled infinitely often. However, x is never executed because it is never enabled. Furthermore, right before each instance of y is executed the processes in $(P_x - P_z) \cup P_y$ are ready, and when y is executed the processes in P_z immediately become ready. Consider the run ρ obtained from $r(S, A)$ by deferring each execution of y until the processes in $P_z - P_y$ become ready. Clearly, $\rho \rightsquigarrow r(S, A)$. Moreover, each deferment causes x to be enabled right before y is executed. So x is enabled infinitely often in ρ. Since x is never executed, ρ does not satisfy SIF. ■

According to the above theorem, if interactions can involve only a single process, then the smallest system for which SIF is not possible consists of only two processes p_1 and p_2 and three interactions x,

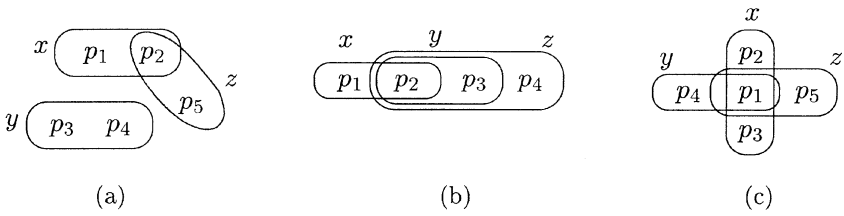

FIG. 5. Interaction structures for which SIF is possible.

y, and z such that $P_y = \{p_1\}$, $P_z = \{p_2\}$, and $P_x = \{p_1, p_2\}$ (see Fig. 4a). Otherwise, the one shown in Fig. 4b would be the smallest. In either case, SIF is not possible for both biparty and multiparty interactions.

Conversely, SIF can be implemented for systems where either no two interactions y and z conflict with a third interaction x, or if they do then it must be the case that $P_y \cap P_x \subseteq P_z$ or $P_z \cap P_x \subseteq P_y$ (see Fig. 5 for some examples).

THEOREM 5.2. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^{\forall})$. *Assume* (1) $|\mathsf{I}| \leq 2$ *or* (2) $\forall x, y, z \in \mathsf{I}$, *if* $P_x \cap P_y \neq \emptyset$ *and* $P_x \cap P_z \neq \emptyset$; *then either* $P_x \cap P_y \subseteq P_z$ *or* $P_x \cap P_z \subseteq P_y$. *Then SIF can be implemented for* $\mathbb{IS}$.

Proof. We present a nonblocking scheduler S satisfying SIF in Fig. 6. (Note that this implies that SIF is strongly feasible.) The scheduler is based on the schedulers presented in [4, 40], except that we do not need the function of random assignments that renders the scheduler's behavior nondeterministic.

For each interaction x, the scheduler S associates with x a unique id and maintains a variable $x.count$ (initialized to zero) recording the number of rounds in which x is enabled but is not selected for execution. Then, in each round S increments the count variable of each enabled interaction by one. The enabled interaction with the largest count is selected for execution, and its *count* is reset to zero. Tie is broken by, say, selecting the one with the largest interaction id.

We claim that for any given adversary A, the scheduler guarantees the following assertion at the end of every round:

$$\text{INV} = \forall\, 0 \leq k \leq |\mathsf{I}|, |\{x \in \mathsf{I} \mid x.count \geq k\}| \leq |\mathsf{I}| - k.$$

The assertion implies that for every $x \in \mathsf{I}$, $x.count \leq |\mathsf{I}|$, which then implies that $r(S, A)$ satisfies SIF. It is easy to see that INV holds initially as $x.count$ is initialized to 0. Assume that INV holds at the end of round i. By contradiction, if INV does not hold at the end of round $i+1$, then there must exist some k, $k \geq 1$, such that $|\{x \in \mathsf{I} \mid x.count \geq k\}| \geq |\mathsf{I}| - k + 1$. Let $Z = \{x \in \mathsf{I} \mid x.count \geq k\}$ be the set of interactions whose counts are at least k at the end of round $i+1$. By the definition of S, for each $x \in Z$, $x.count \geq k-1$ at the end of round i. Therefore, there are at least $|Z|$ interactions having counts at least $k-1$ at the end of round i. By the induction hypothesis, however, at most $|\mathsf{I}| - k + 1$ interactions can have their counts greater than or equal to $k-1$; that is, $|Z| \leq |\mathsf{I}| - k + 1$. So $|Z| = |\mathsf{I}| - k + 1$, and the set of interactions whose counts are at least $k-1$ at the end of round i is equal to Z. Since INV holds at the end of round i but not at the end of round $i+1$, there must exist some interaction $y \in Z$ such that $y.count$ has been changed in round $i+1$. This means that y is enabled prior to round $i+1$. By the definition of S, some interaction will be chosen for execution, and the interaction (say y') must be in Z. Once y' is executed its count is reset to 0. However, this contradicts the fact that $y'.count \geq k > 0$ at the end of round $i+1$.

```
in each round do
    for each enabled interaction x increment its count variable by 1;
    select for execution an enabled interaction with the maximal value
        for its count variable; tie is broken by interaction id;
    reset the count variable of the selected interaction to 0
```

FIG. 6. A nonblocking scheduler for SIF.

We now show that if the interaction structure of $\mathbb{IS}$ satisfies the conditions stated in the theorem, then for every adversary A all runs in $indistinct(r(S, A))$ must satisfy SIF. By Theorem 3.2, there is a scheduling program for $\mathbb{IS}$ satisfying SIF. Suppose otherwise some run in $indistinct(r(S, A))$ does not satisfy SIF. Then, by the discussion presented in the beginning of this section, there must exist three interactions x, y, and z such that $P_x \cap P_z \not\subseteq P_y$ and $P_x \cap P_y \not\subseteq P_z$. This then contradicts the assumption imposed on the structure of I. ■

In the above we assumed that the programs of interaction systems are of type $\mathsf{M}^\forall$. One would also be interested to know, given any $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ where M is not limited to type $\mathsf{M}^\forall$, whether SIF is implementable for $\mathbb{IS}$. In this case, the implementability is determined not only by the structure of I but also by the condition whether the semantics of M allows the interactions to be enabled as required so as to intrigue against SIF. So the structure of I presented in Theorem 5.1 becomes only a necessary condition for the unimplementability. It should be noted again that when a process does not need to be ready for all interactions of which it is a member at a time, then the condition $P_x \cap P_y \not\subseteq P_z$ may be lifted from Theorem 5.1. That is, we only require I to contain three interactions x, y, and z such that $P_x \cap P_y \neq \emptyset$ and $P_x \cap P_z \not\subseteq P_y$. This can be seen from the discussion in the beginning of this section and from the following example: Let $\mathbb{IS} = (\{p_1, p_2\}, \{x, y, z\}, \mathsf{M})$ be an interaction system such that $P_x = P_z = \{p_1, p_2\}$, $P_y = \{p_1\}$, and M behaves as follows:

$$p_1 :: *[x \to \texttt{skip} \quad \square\, y \to z] \qquad p_2 :: *[x \to \texttt{skip} \quad \square\, z \to \texttt{skip}]$$

Here we use the CSP notion $*[\cdots]$ to represent a repetitive command. Then run($\mathbb{IS}$) contains a run $\pi = (p_1 y p_1.\{z\} p_2 z)^\omega$ satisfying SIF. Since π is singular and $indistinct(\pi)$ contains a non-SIF run $(p_1 p_2 y p_1.\{z\} z)^\omega$, by Lemma 4.3 SIF is not implementable for the system.

In addition to the above structure requirement, from Theorems 3.2 and 5.1, it can also be seen that if the semantics of M allows that from some state s onward all continuations of s drive $\mathbb{IS}$ either into a terminating state or into a state in which x and y can be enabled simultaneously, and after y is executed z can subsequently be executed without ever requiring x to be enabled; then SIF is still unimplementable if s has an infinite continuation and all nonblocking schedulers inevitably drive $\mathbb{IS}$ into state s.

To illustrate, consider the program

$$p_1 :: *[x \to \texttt{skip} \quad \square\, z \to \texttt{skip}] \qquad p_2 :: *[x \to \texttt{skip} \quad \square\, y \to \texttt{skip}] \qquad p_3 :: *[y \to z]$$

where x, y, and z are interactions as depicted in Fig. 4b. Clearly, the program does not belong to the category of $\mathsf{M}^\forall$, as p_3 is ready for y and z alternately. However, the program still allows a conspiracy to be constructed so that from the time when y is executed onward, the rest of the run is such that z and y alternately become enabled and executed (i.e., the run is like this: $\ldots (p_1 p_3.\{z\} z p_2 p_3.\{y\} y)^\omega$), precluding x from ever being enabled. Such a run then has an indistinguishable run in which x is enabled infinitely often but is executed only a finite number of times. Since every nonblocking scheduler satisfying SIF for such a system must eventually schedule y to be executed, SIF cannot be implemented for the system. Note that the adversary takes an advantage of the fact that although x and y can be enabled simultaneously, it delays the readiness of the processes in $P_x - P_y$ until y is executed, and then let z be the only choice of the processes in $P_x - P_y$.

On the other hand, by a proof similar to Theorem 5.2, we can show that if the semantics of M does not allow the three interactions x, y, and z to be enabled in a desirable way as described above, then SIF can be implemented. For example, consider a variation of the above program:

$$p_1 :: *[x \to \texttt{skip} \quad \square\, z \to \texttt{skip}] \qquad p_2 :: *[x \to y] \qquad p_3 :: *[y \to z]$$

In this program x and y can never be enabled simultaneously, and so no conspiracy like the above can be constructed to prevent x from being enabled but not executed. Note that by switching the roles of y and z, we see that x and z can be enabled simultaneously. But still, no conspiracy against x can be

constructed because a continuation from a state in which both x and z are enabled but z is chosen for execution always leads to a state in which x is the only choice for execution. (It can also be seen that there is no conspiracy against y and z, either.)

We are now left with the case where some states can lead to a conspiracy against SIF, but some cannot. By Theorem 3.2, the implementability then depends on whether there exists a nonblocking scheduler that not only satisfies SIF but can also prevent itself from "painting into a corner" by driving the system into a state from which a conspiracy against SIF cannot be avoided.

As an example, consider a system $\mathbb{S} = (\{p_1, p_2, p_3\}, \{x, y, z, u\}, \mathsf{M})$, where x, y, z are structured as in Fig. 4b, $P_u = \{p_2\}$, and M behaves as follows:

$p_1 :: *[x \to \texttt{skip}$
$\quad \Box\, z \to \texttt{skip}]$

$p_2 :: i := 0; j := 0;$
$\quad *[i + j < 10; x \to i := i + 1$
$\quad\ \Box\, i + j < 10; y \to j := j + 1$
$\quad\ \Box\, i + j < 10; u \to j := j + 1];$
$\quad \texttt{if}\ i \geq j\ \texttt{then}$
$\quad\quad *[x \to \texttt{skip}$
$\quad\quad\ \Box\, y \to \texttt{skip}]$
$\quad \texttt{else}\ *[x \to y]$

$p_3 :: *[y \to z]$

The program first lets p_2 execute 10 interactions and then execute either the repetitive command $*[x \to \texttt{skip} \,\Box\, y \to \texttt{skip}]$ or the command $*[x \to y]$, depending on how may times x, y, and u have been executed in the first 10 interactions. By the previous two examples, the implementability of SIF then depends on which repetitive command is executed, i.e., depends on which of the two states $i \geq j$ and $i < j$ (when $i + j = 10$) is reached. We can easily design a nonblocking scheduler to ensure that no matter which adversary is given, only the state $i < j$ can be reached. The scheduler behaves like the one presented in Fig. 6, except that for the first 10 states where p_2 is ready for interaction, p_2 always executes u.

On the other hand, if we remove the guarded command "$i + j < 10; u \to j := j + 1$" from p_2's program, then some adversary would be able to drive the system into the state $i \geq j$ regardless of which nonblocking scheduler is employed. To do so, the adversary simply lets x be the only interaction enabled at a time until 10 instances of x are executed. (From then on, the adversary can then contrive a conspiracy against SIF.) So SIF becomes unimplementable for the new system.

In the above examples, determining SIF's implementability reduces to the problem of determining whether some state of the system is reachable. Unfortunately, state reachability, like the *halting problem*, is in general undecidable. Therefore, in the worst case determining whether SIF is implementable or not for a given system is also undecidable!

52. Strong Process Fairness

The notion of strong process fairness requires a process that is infinitely often ready for an enabled interaction to participate in an interaction infinitely often. Unfortunately, like SIF, SPF is, in general, impossible to implement. To see this, observe first that if π satisfies SPF, then all runs in $permute(\pi)$ satisfy SPF. So if π satisfies SPF but some run indistinguishable from π does not, then there must exist some $\pi' \in permute(\pi)$ such that π' contains infinitely many sequences of the form

$$yq_1q_2 \ldots q_k$$

such that $k \geq 1$, $q_1, q_2, \ldots q_k \notin P_y$, and deferring y until q_k causes some interaction x (which has been enabled only a finite number of times in π') to be enabled immediately before y is executed. So $P_x \cap P_y \neq \emptyset$ and $P_x - P_y \neq \emptyset$. Moreover, there must exist some process in $P_x - P_y$ (say p_1) which at some point in π' is ready for interaction and remains ready thereafter because no interaction involving p_1 will be enabled. There must also exist a second process (say p_2) in $P_x - P_y$ such that p_2 will execute some interaction z after the sequence $yq_1q_2 \ldots q_k$ (for otherwise when two sequences

of the form $yq_1q_2 \ldots q_k \ldots$ are placed next to each other, x would be immediately enabled before y is executed). So $P_x \cap P_z \not\subseteq P_y$. Let z be the first such interaction that is executed after the sequence $yq_1q_2 \ldots q_k$. Clearly, $p_1 \notin P_z$; that is, $P_x - P_y - P_z \neq \emptyset$.

Taking z into account, π' contains infinitely many sequences of the form $yq_1q_2 \ldots q_k \ldots z \ldots$. Since prior to the execution of z all processes in $P_x - P_y$ are ready, $P_x \cap P_y$ cannot be all ready before z is executed. Otherwise, x would be enabled immediately before z is executed, and so would be enabled infinitely often throughout π'. So there must exist a third process in $P_x \cap P_y$ which is not ready immediately before z is executed; that is, $P_x \cap P_y \not\subseteq P_z$.[8]

Finally, since no interaction involving p_1 can be enabled infinitely often, the structure of I must guarantee that at any point of the sequence $yq_1q_2 \ldots q_k \ldots z \ldots$ no interaction involving p_1 can be enabled while some of the processes $P_x \cup P_y \cup P_z$ are ready.[9]

Hence, if the interaction structure of the underlying system satisfies the above conditions and a run like π is inevitable to every nonblocking scheduler, then SPF will be impossible for the system. To illustrate such a setting, consider an interaction system $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$, where I has the structure as depicted in Fig. 7c. Then the following are two runs of the system:

$$\pi = p_1(p_3p_5yp_2p_4z)^\omega$$
$$\rho = p_1(p_3p_5p_2yp_4z)^\omega.$$

Observe that π satisfies SPF and $\rho \rightsquigarrow \pi$. However, ρ does not satisfy SPF because p_1 is ready for x each time when both p_2 and p_3 are ready but p_1 never takes part in any interaction execution. Since π is singular, by Lemma 4.3 SPF is not possible for the system.

THEOREM 5.3. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$. *Assume there are* $x, y, z \in \mathsf{I}$ *satisfying the following conditions*:

1. $P_x - P_y - P_z \neq \emptyset$, $P_x \cap P_y \not\subseteq P_z$, *and* $P_x \cap P_z \not\subseteq P_y$.
2. $\exists\, p_1 \in P_x - P_y - P_z, \forall u \in \mathsf{I}, p_1 \in P_u \Rightarrow [P_u \not\subseteq (P_x - P_y) \cup P_z \text{ and } P_u \not\subseteq (P_x - P_z) \cup P_y]$.

Then, SPF cannot be implemented for $\mathbb{IS}$.

Proof. Let S be any given nonblocking scheduler satisfying SPF. We present an adversary A such that some run in *indistinct*$(r(S, A))$ does not satisfy SPF. By Theorem 3.2, therefore, there is no scheduling program for $\mathbb{IS}$ satisfying SPF. The technique in constructing the adversary is similar to Theorem 5.1. However, finding appropriate interactions and processes for which the adversary can be constructed is somewhat tedious. To illustrate the main idea of the construction, we shall first make an additional assumption on $\mathbb{IS}$:

3. $\exists\, p_2 \in P_z - P_y, \exists\, p_3 \in P_y - P_z, \forall\, u \in \mathsf{I}$, $[[p_2 \in u \text{ and } u \subseteq (P_x - P_y) \cup P_z \Rightarrow P_z \subseteq P_u]]$ and $[[p_3 \in u \text{ and } u \subseteq (P_x - P_z) \cup P_y] \Rightarrow P_y \subseteq P_u]$.

Later we sketch the proof without this extra condition.

Let x, y, z, p_1, p_2, and p_3 be as given in Conditions 1–3. The adversary is given as follows.

1. Initially, the adversary A schedules all processes in $(P_x - P_z) \cup P_y$ to enter a ready state (in arbitrary order). By the definition of $\mathsf{M}^\forall$, p_3 is ready for an enabled interaction (because y and possibly some other interactions are enabled), but p_2 is idle (because $p_2 \in P_z - P_y$). Also, p_1 is ready but no interaction involving it is enabled due to the second part of Condition 2 imposed on the structure of I.

2. A's subsequent behavior then depends on S's reaction.

2.1. If S selects an interaction y' (not necessary y) involving p_3, then A in turn schedules the processes in P_z to enter a ready state. By Condition 3, $P_y \subseteq P_{y'}$. So at this point the processes in $(P_x - P_{y'}) \cup P_z$ are ready for interaction. So p_2 is ready for an enabled interaction (z and possibly

[8] Like SIF, the condition $P_x \cap P_y \not\subseteq P_z$ is needed only because the definition of $\mathsf{M}^\forall$ lets all processes in $P_x \cap P_y$ be ready to execute x whenever they are ready. Irrespective of M and $\mathsf{M}^\forall$, however, P_x must contain at least three processes in order to contrive a conspiracy against SPF. This can be seen from the above conditions that $P_x \cap P_y \neq \emptyset$, $p_1 \neq p_2$, and $p_1, p_2 \in P_x - P_y$.

[9] Note again that the condition that restrains I from containing any interaction u that may cause p_1 to be ready for an enabled interaction infinitely often is needed only because the program in consideration is of type $\mathsf{M}^\forall$. Otherwise, the program can take the role to prevent p_1 from being ready for any enabled interaction that may avoid the conspiracy.

some others), but p_3 is idle. Also, p_1 remains ready and, by Condition 2, no interaction involving p_1 is enabled at this point.

2.1.1. If S next selects an interaction z' involving p_2, then A schedules the idle processes in $P_{y'}$ to be ready and waits for S's response. By Condition 3, $P_z \subseteq P_{z'}$. So at this point the processes in $(P_x - P_{z'}) \cup P_y$ are ready for interaction. So p_3 is again ready for an enabled interaction (e.g., y), p_2 is idle, and p_1 remains ready. By Condition 2 no interaction involving p_1 can be enabled at this point. The following behavior of A is same as the beginning of step 2 (except that after S has selected some y' involving p_3, A in turn schedules the idle processes in $P_{z'}$ to be ready).

2.1.2. If, however, S selects some interaction v which does not involve p_2, then A in response schedules the processes in P_v to enter a ready state so that p_2 is again ready for an enabled interaction in S's next turn. If subsequently S chooses some interaction z' involving p_2, then A schedules the idle processes in $P_{y'}$ to enter a ready state as described in step 2.1.1. Otherwise, A continues to schedule the set of processes for which S has just selected for interaction to enter a ready state. Note that in this tournament some interaction involving p_2 will eventually be chosen because S satisfies SPF. Meanwhile, p_1 remains ready but still no interaction involving it has been enabled.

2.2. If S selects some interaction u which does not involve p_3, then A in response schedules the processes in P_u to enter a ready state so that p_3 is again ready for an enabled interaction (e.g., y) in S's turn. If S finally chooses some interaction y' involving p_3, then A behaves as that described in step 2.1. Otherwise, A continues to schedule the set of processes for which S has just selected for interaction to become ready. As discussed above, in this tournament some interaction involving p_3 will eventually be selected because S satisfies SPF, while p_1 remains ready but no interaction involving it has been enabled.

Then S versus A must generate a run in which p_3 executes some interaction (say y') infinitely often because it is ready for an enabled interaction infinitely often. However, p_1 is ready forever but it never executes an interaction because it is never involved in an enabled interaction. Furthermore, immediately before p_3 executes each instance of y' the processes in $(P_x - P_{z'}) \cup P_y$ are ready, and after p_3 executes y' the processes in $P_{z'}$ are ready before any interaction is to be executed. Consider the run ρ obtained from $r(S, A)$ by deferring each execution of y' until the processes in $P_x - P_{y'}$ are all ready. Clearly, $\rho \rightsquigarrow r(S, A)$. Moreover, each deferment causes x to be enabled right before y' is executed. Then, p_1 is ready for an enabled interaction (i.e., x) infinitely often in ρ. Since x is never executed, ρ does not satisfy SPF.

We now consider the theorem without Condition 3. The main idea behind the construction of the adversary is the same: it chooses three processes p_1, p_2, and p_3 in an interaction x and lets p_2 and p_3 execute interactions alternately, but prevents p_1 from being involved in any enabled interaction during the game versus S. Without Condition 3, however, not every set of x, y, and z (where y and z are used to determine p_3 and p_2) satisfying Conditions 1 and 2 can be used to contrive a conspiracy, as some may cause p_1 to be exposed to an enabled interaction during the game. In the following we show how to obtain p_2 and p_3 for the adversary. Once they are obtained, the proof of the theorem is essentially the same as above, and so we shall omit the details. (Note that arguing that no interaction involving p_1 can be enabled at any point during the game is quite tedious, but is not hard.)

Let x, y, z, and p_1 be as given in the theorem statement. Without loss of generality, assume that x, y, and z satisfy the following condition:

I. $|P_x \cup P_y \cup P_z|$ is minimal; that is, there are no other x', y' and z' satisfying Conditions 1 and 2 such that $|P_{x'} \cup P_{y'} \cup P_{z'}| < |P_x \cup P_y \cup P_z|$.

Define $\mathsf{I}_{p;x,y,z} = \{u \in \mathsf{I} \mid P_u \subseteq (P_x - P_z) \cup P_y\}$, and $rank(p; x, y, z) = \min\{|P_u \cap (P_x \cap P_y - P_z)| \mid u \in \mathsf{I}_{p;x,y,z}\}$. Assume that in the latest two rounds S schedules z' in Step 2.1.1 and schedules y' in Step 2.1. (Initially, $z' = z$ and $y' = y$). Then in the new round "p_2" and "p_3" are dynamically determined as follows: "p_3" in Step 2.1 is chosen to be the process in $P_x \cap P_{y'} - P_{z'}$ with the maximum $rank(p; x, y', z')$, and "p_2" in Step 2.1.1 is chosen to be the process in $P_x \cap P_{z'} - P_{y'}$ with the maximum $rank(p; x, z', y')$.

Figure 7 illustrates some interaction structures for which SPF is not possible. All of them consist of an interaction involving more than two processes. It is proven in [4] that SPF for purely biparty interactions is equivalence robust. SPF is also strongly feasible because SIF is strongly feasible and SPF is weaker

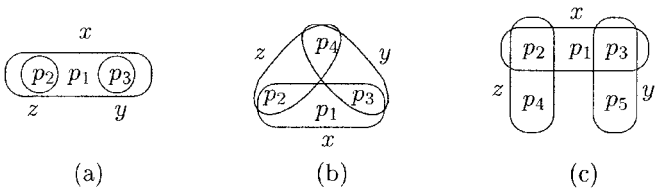

FIG. 7. Interaction structures for which SPF cannot be implemented.

than SIF. By Lemma 4.1, therefore, SPF is implementable for systems consisting of strictly biparty interactions (even if the associated programs are not of type $\mathsf{M}^{\forall}$).

THEOREM 5.4. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^{\forall})$. *Assume that for all* $x, y, z \in \mathsf{I}$ *the following conditions do not hold*:

I. $P_x - P_y - P_z \neq \emptyset$, $P_x \cap P_y \not\subseteq P_z$, *and* $P_x \cap P_z \not\subseteq P_y$, *and*

II. $\exists\, p_1 \in P_x - P_y - P_z, \forall u \in \mathsf{I}, p_1 \in P_u \Rightarrow [P_u \not\subseteq (P_x - P_y) \cup P_z,$ *and* $P_u \not\subseteq (P_x - P_z) \cup P_y]$.

Then, SPF is implementable for $\mathbb{IS}$.

Proof. Since every run that satisfies SIF must satisfy SPF, the nonblocking scheduler S presented in Fig. 6 also satisfies SPF. So to show that SPF is implementable for the system, by Theorem 3.2 it suffices to show that for every adversary A all runs in *indistinct*$(r(S, A))$ satisfy SPF. Let $\pi = r(S, A)$. Suppose by contradiction that π satisfies SPF but some run in *indistinct*(π) does not. By the argument presented in the beginning of this section, there must exist some $\pi' \in$ *permute*(π) and three interactions x, y, z, where $P_x - P_y - P_z \neq \emptyset$, $P_x \cap P_y \not\subseteq P_z$, and $P_x \cap P_z \not\subseteq P_y$, such that π' contains infinitely many sequences of the form

$$yq_1q_2 \ldots q_k \ldots z$$

but moving $q_1q_2 \ldots q_k$ ahead of y causes x to be enabled immediately before y is executed. Here z is the first interaction executed after y that involves a process in $P_x - P_y$. Moreover, there must exist some $p_1 \in P_x - P_y - P_z$ such that from some point onward, p_1 remains ready forever in π' because no interaction involving p_1 is enabled.

Since before z is executed the set of processes $(P_x - P_y) \cup P_z$ are ready, and since no interaction involving p_1 is enabled at this point, I contains no u such that $p_1 \in P_u$ and $P_u \subseteq (P_x - P_y) \cup P_z$. If there is also no u such that $p_1 \in P_u$ and $P_u \subseteq (P_x - P_z) \cup P_y$, then we have obtained three interactions x, y, z and a process $p_1 \in P_x$ that contradict the theorem assumption imposed on the structure of I. So in the following we assume that there is some u such that $p_1 \in P_u$ and $P_u \subseteq (P_x - P_z) \cup P_y$. We shall show that this assumption leads to a contradiction, and so there is no π such that π satisfies SPF but some run in *indistinct*(π) does not. Without loss of generality assume that $|P_u \cup P_y|$ is the "smallest" in the sense that there is no other interaction u' such that $p_1 \in P_{u'}$, $P_{u'} \subseteq (P_x - P_z) \cup P_y$, and $|P_{u'} \cup P_y| < |P_u \cup P_y|$.

When two sequences of the form $yq_1q_2 \ldots q_k \ldots z$ are placed next to each other, we have a sequence

$$yq_1q_2 \ldots q_k \ldots z \ldots yq_1q_2 \ldots q_k \ldots z \ldots .$$

So the set of processes $(P_x - P_z) \cup P_y$ are ready immediately before the second instance of y is executed (and so u is enabled immediately before that y is executed), unless an interaction z' which does not involve p_1 but does involve some process in $(P_x - P_z - P_y) \cap P_u$ is placed in between z and y. Since u cannot be enabled at any point in the above sequence, such a z' exists; and, in addition, $P_u \cap P_{z'} \not\subseteq P_y$. Let z' be the first such interaction executed in between z and y.

On the other hand, it can also be seen that $P_u \cap P_y \neq \emptyset$ and $P_u \cap P_y \not\subseteq P_{z'}$, for otherwise u would be enabled immediately before z' is executed. Recall that $p_1 \in P_u - P_y - P_{z'}$. So up to this point we have obtained three interactions u, z', y such that $P_u - P_y - P_{z'} \neq \emptyset$, $P_u \cap P_y \not\subseteq P_{z'}$, and $P_u \cap P_{z'} \not\subseteq P_y$.

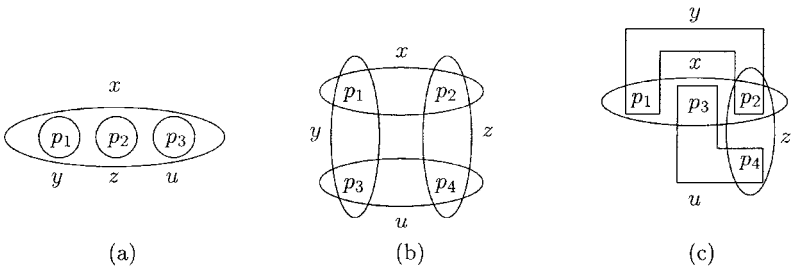

FIG. 8. Interaction structures for which SPF can be implemented.

We now argue that there is no $v \in \mathsf{I}$ such that $p_1 \in P_v$ and $P_v \subseteq (P_u - P_{z'}) \cup P_y$. This is because if such v exists, then since $(P_u \cap P_{z'}) - P_y \neq \emptyset$, we must have $|P_v \cup P_y| < |P_u \cup P_y|$; this then contradicts the assumption we made earlier that no such v exists. Moreover, since z' is the first interaction executed in between z and y satisfying the condition $P_u \cap P_{z'} \neq \emptyset$ and $P_u \cap P_{z'} \not\subseteq P_y$, the set of processes $(P_u - P_y) \cup P_{z'}$ are all ready immediately before z' is executed. So there cannot exist any interaction v such that $p_1 \in P_v$ and $P_v \subseteq (P_u - P_y) \cup P_{z'}$. Then the existence of u, z', and y contradicts the assumption imposed on the structure of I. So there cannot exist an interaction u such that $p_1 \in P_u$ and $P_u \subseteq (P_x - P_z) \cup P_y$. The theorem then is established. ■

Note that since a run satisfying SIF must also satisfies SPF, interaction systems for which SIF is possible can also be implemented for SPF. Therefore, SPF can be implemented for systems with the interaction structures shown in Fig. 5. Figure 8 illustrates some more examples for which SIF is not possible but for which SPF is possible.

By a reasoning similar to the one presented for SIF in Section 5.1, one can also determine the structure of I and the semantics of M rendering the possibility and impossibility phenomena of SPF for any given $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$, where M is not limited to type $\mathsf{M}^\forall$. Note that because some system causes SPF to be unimplementable and some does not, like SIF, we can also construct a system such that determining SPF's implementability reduces to the problem of determining whether some state of the system is reachable. Therefore, in the worst case determining whether SPF is implementable or not for a given system is also undecidable.

53. Weak Process Fairness

The notion of weak process fairness requires a process that is continuously ready for an enabled interaction (not necessary the same interaction) to execute an interaction eventually. Like SIF and SPF, for some interaction system WPF may include a run π such that some run indistinguishable from π does not satisfy WPF. This can be illustrated by the interaction system $\mathbb{IS} = (\{p_1, p_2, p_3, p_4, p_5\}, \{u, v, x, y\}, \mathsf{M}^\forall)$, where $P_u = \{p_1, p_5\}$, $P_v = \{p_2, p_5\}$, $P_x = \{p_1, p_3\}$, and $P_y = \{p_2, p_4\}$ (see Fig. 9). Then the following are two runs of the system:

$$\pi = p_5 p_1 p_3 (x p_2 p_4 y p_1 p_3)^\omega$$

$$\rho = p_5 p_1 p_3 (p_2 p_4 x p_1 p_3 y)^\omega.$$

Observe that $\rho \rightsquigarrow \pi$ and π satisfies WPF because no process is continuously ready for an enabled interaction. However, ρ does not satisfy WPF because from the second state onward p_5 is continuously ready for an enabled interaction (u and v alternately), but it never executes any interaction.

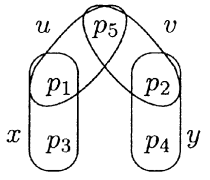

FIG. 9. An interaction structure.

Note that π is not singular. So it is not necessary that every nonblocking scheduler for the system must generate π. So if we can devise a nonblocking scheduler which can avoid generating runs like π whose *indistinct*(π) contains a non-WPF run, then we can obtain an implementation for WPF. In fact, we can show that if *indistinct*(π) contains a non-WPF run, then π must not satisfy SIF. Therefore, if we can construct a nonblocking scheduler which generates only SIF runs, then we will have an implementation for WPF. Since such a scheduler is indeed possible (because SIF is strongly feasible; see Section 5.1), WPF is implementable.

LEMMA 5.5. *For any run $\pi \in run(\mathbb{IS})$, if π satisfies SIF, then all runs in indistinct(π) satisfy WPF.*

Proof. Let $\pi \in \text{run}(\mathbb{IS})$ be a run satisfying SIF. Clearly, π must also satisfy WPF. Moreover, if π is finite or no process in π has stayed in a ready state forever, then every run in *indistinct*(π) must satisfy WPF. So suppose that π is infinite, and at some point in π some process p has entered a ready state and remained ready thereafter, but from that point onward no interaction involving p has been enabled. Consider any subsequence of π which is of the form

$$y\ p_1.I_1\ p_2.I_2 \ldots p_k.I_k\ z$$

and assume that in the subsequence p is ready but no interaction involving p is enabled. Let ρ be any arbitrary run in *indistinct*(π). Clearly, if ρ differs from π only in the ordering of the actions $p_1.I_1, p_2.I_2, \ldots, p_k.I_k$ in between y and z, then ρ must also satisfy WPF. This is because the reordering of these actions cannot cause any new interaction to be enabled.

On the other hand, suppose that ρ is obtained from π by moving some of the $p_i.I_i$'s forward before y (via an operation of retraction), and the movement causes some interaction x involving p to be enabled immediately before y is executed. Then, $P_x \cap P_y \neq \emptyset$. So x must be disabled immediately after y is executed. Furthermore, since in the process of retraction any state transition occurring after z cannot be moved forward across y, the duration of p's readiness for an enabled interaction cannot be extended across y. So p cannot be continuously ready for an enabled interaction in ρ. Hence, ρ must also satisfy WPF. ■

THEOREM 5.6. *WPF is implementable for every interaction system $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$.*

Proof. For any given interaction system $\mathbb{IS}$, every run π generated by the the nonblocking scheduler presented in Fig. 6 satisfies SIF, and thus also satisfies WPF. Moreover, by Lemma 5.5 for every π generated by the scheduler, all runs in *indistinct*(π) satisfy WPF. Hence, by Theorem 3.2 there exists an undelayed scheduling program for $\mathbb{IS}$ satisfying WPF. ■

54. U-Fairness

The notion of U-fairness is first proposed by Back and Kurki-Suonio [8] (called by there a different name, *action justice*) to consider situations where each participant p of an interaction x is willing to execute x every time when it is ready for interaction. Subsequently, Attie *et al.* [5] show that U-fairness provides an abstraction for stable property detection which most well-known fairness notions do not.

DEFINITION 5.1 (U-fairness [5]). A run $\pi \in \text{run}(\mathbb{IS})$ satisfies *U-fairness* iff for every interaction x, x will eventually be executed if the following condition is satisfied: if from some point onward every participant p of x is ready for interaction infinitely often, and whenever it is ready for interaction, it is willing to execute x.

Note that in the above definition x may never be enabled because the participants need not be in a ready state simultaneously.

To see how U-fairness detects stable properties, assume that we are to compute a function $g(v_1, v_2)$. The result is acceptable only if $f_1(v_1) \leq \delta_1$ and $f_2(v_2) \leq \delta_2$ for some constants δ_i's and functions f_i's, $i = 1, 2$. Moreover, the smaller $f_i(v_i)$ is, the closer $g(v_1, v_2)$ approaches to the optimal. To do so, we can use two processes p_1 and p_2 to prepare the appropriate v_1 and v_2 respectively, and then let, say, p_1

compute $g(v_1, v_2)$, as shown in the following program:

$$
\begin{aligned}
p_1 :: \; & v_1 := 0;\ a_1 := 0; d_1 := \infty; \mathit{continue}_1 := \texttt{true};\\
& *[\mathit{continue}_1 \,\&\, x[\texttt{if}\ f_1(a_1) < d_1\ \texttt{then}\ d_1 := f_1(a_1); v_1 := a_1;\ \texttt{endif}] \to a_1 := a_1 + 1;\\
& \Box\, \mathit{continue}_1 \wedge f_1(v_1) \le \delta_1 \,\&\, z[\texttt{output}\ g(v_1, v_2); \mathit{continue}_1 := \texttt{false};] \to \texttt{skip};]\\
p_2 :: \; & v_2 := 0; a_2 := 0; d_2 := \infty; \mathit{continue}_2 := \texttt{true};\\
& *[\mathit{continue}_2 \,\&\, y[\texttt{if}\ f_2(a_2) < d_2\ \texttt{then}\ d_2 := f_2(a_2); v_2 := a_2;\ \texttt{endif}] \to a_2 := a_2 + 1;\\
& \Box\, \mathit{continue}_2 \wedge f_2(v_2) \le \delta_2 \,\&\, z[\mathit{continue}_2 := \texttt{false};] \to \texttt{skip};]
\end{aligned}
$$

The program is written in the style of IP [18], where $x[\ldots]$ represents an interaction with name x and body $\ldots$, & denotes the guard operator, and $*[\ldots]$ represents a repetitive command.

Note that since smaller $f_i(v_i)$ implies better results, instead of idling p_i when the threshold $f_i(v_i) \le \delta_i$ is met, we allow p_i to continue to find a better value of v_i while its partner is still preparing the other v-value.

We observe that for each i, the boolean guard $f_i(v_i) \le \delta_i$ of interaction z is a stable property: once it holds, it continues to hold during the computation. So when both processes have found appropriate values for v_1 and v_2, each process is willing to execute z every time when it is ready for interaction. So under U-fairness z is guaranteed to be executed and the result $g(v_1, v_2)$ is then obtained.

On the other hand, z may not necessarily be executed under, say, SIF. For example, assume that the threshold $f_i(v_i) \le \delta_i$ for both $i = 1, 2$ have already been met. Consider the run

$$(p_1 x p_2 y)^\omega,$$

which represents that p_1 becomes ready for interaction, then it executes an instance of x to compute a new value of v_1, and before p_1 finishes x and becomes ready again for interaction, p_2 becomes ready and then executes y to prepare v_2, and so on. The overall computation satisfies SIF because z is never enabled throughout the computation. Note that the run does not satisfy U-fairness.

Although with respect to the above specific example U-fairness is stronger than SIF, as shown in [5, 8], U-fairness is actually incomparable with SIF; that is, a run satisfying SIF does not necessarily satisfy U-fairness, and vice versa. To see an example of a run that satisfies U-fairness but not SIF, consider an interaction system with a structure depicted in Fig. 4c. Assume the associated program allows the run

$$\rho = ((p_1 p_3 p_2\, y\ p_4\, z)(p_1.\{y\}\ p_3\, y\ p_2 p_4\, z))^\omega.$$

Then ρ satisfies U-fairness because p_1 is not always willing to execute x every time when it is ready for interaction. The run does not satisfy SIF because x is infinitely often enabled but is never executed. From this example, it is not difficult to see that for any given $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$, if $\mathrm{U}(\mathbb{IS}) - \mathrm{SIF}(\mathbb{IS}) \neq \emptyset$, then the program M must let some process ready some interaction intermittently, where $\mathrm{U}(\mathbb{IS})$ and $\mathrm{SIF}(\mathbb{IS})$ denote the set of complete runs of $\mathbb{IS}$ satisfying U-fairness and SIF, respectively.

Conversely, if no interaction is readied by a process in an intermittent fashion, then $\mathrm{U}(\mathbb{IS}) - \mathrm{SIF}(\mathbb{IS}) = \emptyset$; that is, U-fairness is either stronger than SIF or equal to SIF. This is proved in the following lemma.

LEMMA 5.7. *For every* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$, $\mathrm{U}(\mathbb{IS}) \subseteq \mathrm{SIF}(\mathbb{IS})$. *Moreover,* $\mathrm{U}(\mathbb{IS}) \neq \mathrm{SIF}(\mathbb{IS})$ *iff* $\exists\, x, y, z, \in \mathsf{I}$, $P_x \cap P_y \not\subseteq P_z$ *and* $P_x \cap P_z \not\subseteq P_y$.

Proof. We first show that $\mathrm{U}(\mathbb{IS}) \subseteq \mathrm{SIF}(\mathbb{IS})$; that is, if a run $\pi \in \mathrm{run}\ (\mathbb{IS})$ is not in $\mathrm{SIF}(\mathbb{IS})$ then it is not in $\mathrm{U}(\mathbb{IS})$, either. To see this, observe that if $\pi \notin \mathrm{SIF}(\mathbb{IS})$, there must exist an interaction x in π such that from some point t onward x is infinitely often enabled but is never executed. Recall that program $\mathsf{M}^\forall$ allows a process, whenever it is ready, to be ready for all interactions of which it is a member. So from t onward, every process $p \in P_x$ is ready for interaction infinitely often, and whenever it is ready it is willing to execute x. However, since x is never executed in π, π does not satisfy U-fairness.

We now show that if $\mathrm{U}(\mathbb{IS}) \subsetneq \mathrm{SIF}(\mathbb{IS})$, then $\exists\, x, y, z \in \mathsf{I}$, $P_x \cap P_y \not\subseteq P_z$ and $P_x \cap P_z \not\subseteq P_y$. Let $\pi \in \mathrm{SIF}(\mathbb{IS}) - \mathrm{U}(\mathbb{IS})$. Then, there exists an interaction x in π such that from some point onward every

$p \in P_x$ is ready for interaction infinitely often, and whenever it is ready for interaction it is willing to execute x. However, the processes of P_x are never ready for x simultaneously (so x is never enabled and thus is never executed). Hence there must exist two processes $p_1, p_2 \in P_x$ such that the following scenario occurs infinitely often: when p_1 is ready, p_2 is idle. Then, after p_1 participates in some interaction y, p_2 becomes ready. Moreover, p_1 remains idle until p_2 participates in another interaction z. Clearly, the conditions $P_x \cap P_y \not\subseteq P_z$ and $P_x \cap P_z \not\subseteq P_y$ are satisfied by the three interactions x, y, and z.

Finally, we show that if $\exists x, y, z \in \mathsf{I}$, $P_x \cap P_y \not\subseteq P_z$ and $P_x \cap P_z \not\subseteq P_y$, then $\mathrm{U}(\mathbb{IS}) \subsetneq \mathrm{SIF}(\mathbb{IS})$. For this, we need to construct a computation π such that $\pi \in \mathrm{SIF}(\mathbb{IS}) - \mathrm{U}(\mathbb{IS})$. Consider first the run

$$\pi = (P_x - P_y - P_z)(P_y\, y\, P_z\, z)^\omega.$$

Here we liberally use set P in a run to represent an arbitrary permutation of the elements in P. Since each $p \in P_x$ is ready for interaction infinitely often, and since the program $\mathsf{M}^\forall$ allows p to be ready for all interactions of which it is a member whenever p is ready, each process in P_x is ready for interaction infinitely often, and whenever it is ready for interaction it is willing to execute x. However, since x is never executed in π, $\pi \notin \mathrm{U}(\mathbb{IS})$. So if $\pi \in \mathrm{SIF}(\mathbb{IS})$, then we are done. Otherwise, there must exist some interaction w such that w is enabled infinitely often in π but w is never executed. Obviously, $w \neq y$ and $w \neq z$. Furthermore, due to the restriction imposed on x, y, and z, $w \neq x$, either. Then there are only two possibilities: either $P_w \subseteq (P_x - P_y - P_z) \cup P_y$ or $P_w \subseteq (P_x - P_y - P_z) \cup P_z$.

Consider the first case, and let π' be as follows:

$$\pi' = (P_x - P_y - P_z)(P_y w P_w y P_z z)^\omega.$$

Note that π' is still not in $\mathrm{U}(\mathbb{IS})$ because for every process in P_x, it is still ready for interaction infinitely often, and whenever it is ready for interaction it is willing to execute x. However, w is executed infinitely often in π'. So if π' is in $\mathrm{SIF}(\mathbb{IS})$, then we are done. Otherwise, there must be another interaction u, $u \neq x, y, z, w$, such that u is enabled infinitely often in π' but it is never executed. Then we can use the same method to convert π' to another run π'' such that u, w, y, and z are executed infinitely often in π'', and still $\pi'' \notin \mathrm{U}(\mathbb{IS})$. Since there are at most a finite number of interactions in I, eventually we can construct a run in $\mathrm{SIF}(\mathbb{IS}) - \mathrm{U}(\mathbb{IS})$.

The other case where $P_w \subseteq (P_x - P_y - P_z) \cup P_z$ can be handled similarly. The lemma is therefore established. ■

From Lemma 5.7 and Theorems 5.1 and 5.2, we can obtain the interaction structures that render the unimplementability and implementability of U-fairness, respectively.

COROLLARY 5.8. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$. *Assume* $\exists x, y, z \in \mathsf{I}$, $P_x \cap P_y \not\subseteq P_z$ *and* $P_x \cap P_z \not\subseteq P_y$. *Then U-fairness cannot be implemented for* $\mathbb{IS}$.

Proof. Suppose otherwise that U-fairness is implementable for $\mathbb{IS}$. Then by Theorem 3.2 there is a nonblocking scheduler S such that for every adversary A of $\mathbb{IS}$ *indistinct*$(r(S, A)) \subseteq \mathrm{U}(\mathbb{IS})$. By Lemma 5.7, *indistinct*$(r(S, A)) \subseteq \mathrm{SIF}(\mathbb{IS})$. That is, there is a nonblocking scheduler S such that for every adversary A of $\mathbb{IS}$ *indistinct*$(r(S, A)) \subseteq \mathrm{SIF}(\mathbb{IS})$. Then by Theorem 3.2 SIF is implementable for $\mathbb{IS}$. This contradicts Theorem 5.1 that SIF is not implementable for $\mathbb{IS}$. ■

COROLLARY 5.9. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$. *Assume* (1) $|\mathsf{I}| \leq 2$ *or* (2) $\forall x, y, z \in \mathsf{I}$, *if* $P_x \cap P_y \neq \emptyset$ *and* $P_x \cap P_z \neq \emptyset$, *then either* $P_x \cap P_y \subseteq P_z$ *or* $P_x \cap P_z \subseteq P_y$. *Then U-fairness can be implemented for* $\mathbb{IS}$.

Proof. By Lemma 5.7 and the restriction imposed on $\mathbb{IS}$ we have $\mathrm{U}(\mathbb{IS}) = \mathrm{SIF}(\mathbb{IS})$. Theorem 5.2 therefore implies that U-fairness is implementable for $\mathbb{IS}$. ■

Note that like SIF, U-fairness is in general impossible to implement even if interactions are strictly bipartied.

For interaction systems $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ whose programs are not of type $\mathsf{M}^\forall$, we can use the method presented in Section 5.1 to analyze how the structure of I and the semantics of M affect the implementability of U-fairness. It is important to note, however, that U-fairness is equivalence-robust; see [18]. This means that for every nonblocking scheduler S satisfying U-fairness, every run π generated by S must satisfy

the condition *indistinct*(π) $\subseteq$ U($\mathbb{IS}$). So the unimplementability must be due to the fact that we cannot even construct a nonblocking scheduler satisfying U-fairness for the system. Recall that we can design a nonblocking scheduler satisfying SIF (see Fig. 6). So for an adversary A to prevent S from generating a U-fair run, I and M must be such that from some point onward some interaction x is never enabled, but every participant of x is ready for interaction infinitely often, and whenever it is ready for interaction it is willing to execute x. Since x is never enabled, there must exist two other interactions y and z, $P_y \cap P_x \not\subseteq P_z$ and $P_z \cap P_x \not\subseteq P_y$, such that y and z alternately engage some of x's participants, preventing the participants of x from being ready for x simultaneously (a phenomenon called *conspiracy*; see Section 5.5). As the proof techniques for establishing the impossibility and possibility results are similar to those for Theorems 5.1 and 5.2, we omit the details.

55. Hyperfairness

Hyperfairness is proposed by Attie *et al.* [6] as a fairness notion to prevent *conspiracies*. A conspiracy against an interaction x occurs if from some point onward x is never enabled because conflicting interactions intermittently engage some of x's participants. For example, consider an interaction system $\mathbb{IS} = (\{p_1, p_2, p_3\}, \{x, y, z\}, \mathsf{M}^{\forall})$, where $\{x, y, z\}$ has the structure as depicted in Fig. 4b. So $\pi = (p_1 p_3 z p_2 p_3 y)^{\omega}$ is a run of the system. Observe that x is never enabled in π because z and y alternately engage p_1 and p_2, respectively. Note that the conspiracy is due solely to the "race conditions" of two independent actions: z's execution and p_2's readiness. If p_2 becomes ready before z is executed, then the resulting computation $\pi' = (p_1 p_3 p_2 z p_3 y)^{\omega}$ would have no conspiracy against x. Hyperfairness is therefore used to exclude π as valid by requiring x_{12} be enabled infinitely often as in π'. Note further that if some other fairness notion, say SIF, is additionally assumed, then π' would not be fair either. Hence, hyperfairness on top of SIF ensures that no computation satisfies SIF simply because some conspiracy prevents an interaction from being enabled (and thus from being potentially scheduled for execution).

Note that some conspiracy may be inherent from the program semantics. For example, assume that the program of the above system is changed to the following:

$$p_1 :: *[x \to \texttt{skip} \quad \square\, z \to \texttt{skip}] \qquad p_2 :: *[y; [x \to \texttt{skip} \quad \square\, y \to \texttt{skip}]] \qquad p_3 :: *[z \to y]$$

Then the program prevents p_1 from establishing x with p_2, unless p_1 first establishes z with p_3.

To distinguish programs for which conspiracies can be prevented by an appropriate scheduling of the execution events from those whose semantics inherently incurs some conspiracies, Attie *et al.* [6] propose a notion of *conspiracy resistance* as follows: A program M is *conspiracy resistant* iff for every fair run π (fair with respect to some underlying fairness notion) the following holds:

> Let π' be any finite prefix of π with final state s, and let Q_x be the set of processes that are ready for x in s. Furthermore, let π'' be the same as π' except that in the final state every process in Q_x is ready for only x. Then for every fair continuation of π'', there exists a process $p \in P_x - Q_x$ such that p will eventually ready x along this continuation.

Hyperfairness excludes conspired runs for conspiracy resistant programs, but does not impose any constraint (other than that imposed by the underlying fairness notion) on programs that are not conspiracy resistant to avoid any possibility of deadlock in the implementation. Formally, hyperfairness is defined as follows.

DEFINITION 5.2 (Hyperfairness [6]). A complete run π of $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ is *hyperfair* iff one of the following conditions is satisfied:

I. M is not conspiracy resistant and π satisfies SIF.

II. π is finite.

III. π is infinite, π satisfies SIF, and every interaction for which every participant readies it infinitely often is enabled infinitely often.

Note that like [6], we have assumed SIF beneath hyperfairness. A different hyperfairness notion would be required if SIF is replaced by another. Its implementability can then be studied analogously.

Since hyperfairness imposes additional constraint on runs that satisfy SIF and since SIF, in general, is impossible to implement, hyperfairness is also unimplementable. What interests us, then, is whether hyperfairness is possible for those cases where SIF is possible. For this problem, we again assume an interaction system $\mathbb{IS}$ associated with a program $\mathsf{M}^\forall$. Clearly, $\mathsf{M}^\forall$ is conspiracy resistant. Moreover, run($\mathbb{IS}$) contains only infinite runs. Thus a run $\pi \in \text{run}(\mathbb{IS})$ is hyperfair iff it satisfies SIF and every interaction that is infinitely often readied by every participant is enabled infinitely often.

To study the structure of interaction systems that renders the implementability and unimplementability phenomena of hyperfairness, we first observe that U-fairness and hyperfairness are indeed the same semantic constraint for those systems associated with $\mathsf{M}^\forall$. This is shown in the following lemma, where Hyper($\mathbb{IS}$) denotes the the set of runs in run($\mathbb{IS}$) satisfying hyperfairness.

LEMMA 5.10. *For every* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$, Hyper($\mathbb{IS}$) = U($\mathbb{IS}$).

Proof. We first show that Hyper($\mathbb{IS}$) $\subseteq$ U($\mathbb{IS}$). Let $\pi \in \text{run}(\mathbb{IS}) - \text{U}(\mathbb{IS})$. Since π is not U-fair, there exists some interaction x such that from some point onward every process in P_x is ready for interaction infinitely often, and whenever it is ready for interaction it is willing to execute x, but x is never executed. Then, x is infinitely often readied by all of its participants but is never executed. So $\pi \notin \text{Hyper}(\mathbb{IS})$. This implies that Hyper($\mathbb{IS}$) $\subseteq$ U($\mathbb{IS}$).

Next, suppose that ρ is U-fair. Then, for every interaction x, x will eventually be executed if from some point onward every process in P_x is ready for interaction infinitely often, and whenever it is ready for interaction it is willing to execute x. Due to the semantics of $\mathsf{M}^\forall$, no interaction that is infinitely often readied by all of its participants is executed only a finite number of times. That is, every interaction that is infinitely often readied by all of its participants is executed infinitely often. This also implies that every interaction that is enabled infinitely often is executed infinitely often. So ρ is also hyperfair. Hence, U($\mathbb{IS}$) $\subseteq$ Hyper($\mathbb{IS}$). ■

The above lemma together with Corollaries 5.8 and 5.9 immediately implies the following two corollaries, respectively.

COROLLARY 5.11. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$. *Assume* $\exists\, x, y, z \in \mathsf{I}$, $P_x \cap P_y \not\subseteq P_z$ *and* $P_x \cap P_z \not\subseteq P_y$. *Then hyperfairness cannot be implemented for* $\mathbb{IS}$.

COROLLARY 5.12. *Let* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M}^\forall)$. *Assume* (1) $|\mathsf{I}| \leq 2$ *or* (2) $\forall x, y, z \in \mathsf{I}$, *if* $P_x \cap P_y \neq \emptyset$ *and* $P_x \cap P_z \neq \emptyset$, *then either* $P_x \cap P_y \subseteq P_z$ *or* $P_x \cap P_z \subseteq P_y$. *Then hyperfairness can be implemented for* $\mathbb{IS}$.

So like SIF and U-fairness, for systems consisting of only biparty interactions hyperfairness is, in general, not implementable.

Using the fairness implementability criterion, we can also analyze the implementability of hyperfairness for any given system $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ whose program is not limited to type $\mathsf{M}^\forall$ (but is conspiracy resistant). We note here that, like U-fairness, hyperfairness is also equivalence-robust [6]. So hyperfairness can pass the criterion if, and only if, we can construct a nonblocking scheduler for the system satisfying hyperfairness.

Finally, it is interesting to note that for every $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ whose M is conspiracy resistant, hyperfairness is, in general, stronger than U-fairness.[10]

THEOREM 5.13. *For every* $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ *such that* M *is conspiracy resistant*, Hyper($\mathbb{IS}$) $\subseteq$ U($\mathbb{IS}$).

Proof. Let $\pi \in \text{run}(\mathbb{IS}) - \text{U}(\mathbb{IS})$. Since π is not U-fair, there exists an interaction x such that from some point onward every process in P_x is ready for interaction infinitely often, and whenever it is ready for interaction it is willing to execute x, but x is never executed. Then, x is infinitely often readied by all of its participants but is never executed. Since M is conspiracy resistant, π should also be excluded from Hyper($\mathbb{IS}$). So Hyper($\mathbb{IS}$) $\subseteq$ U($\mathbb{IS}$). ■

Note that for some interaction systems hyperfairness may be strictly stronger than U-fairness. For example, let $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ be an interaction system with the structure shown in Fig. 4c, and then M

[10] For those $\mathbb{IS} = (\mathsf{P}, \mathsf{I}, \mathsf{M})$ whose M is not conspiracy resistant, hyperfairness on top of SIF imposes no additional constraint other than that imposed by SIF. So hyperfairness is identical to SIF. It is not difficult to see that U-fairness then is still incomparable with SIF (and thus incomparable with hyperfairness).

behaves as follows:

$$p_1 :: *[x \to y \quad \square\, y \to y] \qquad p_2 :: *[x \to \texttt{skip} \quad \square\, z \to \texttt{skip}] \qquad p_3 :: *[y \to \texttt{skip}] \qquad p_4 :: *[z \to \texttt{skip}]$$

It is not difficult to see that the program is conspiracy resistant. Consider the run

$$\rho = ((p_1 p_3 p_2\, y\, p_4\, z)(p_1.\{y\} p_3\, y\, p_2 p_4\, z))^{\omega}.$$

As shown in Section 5.4, ρ satisfies U-fairness but does not satisfy SIF. So ρ is not hyperfair.

6 CONCLUSIONS

We have presented a necessary and sufficient criterion for determining the implementability of fairness notions in distributed systems where processes interact by engaging in synchronous constructs. As we have seen, the criterion allows us to establish several impossibility results for various fairness notions, including strong interaction fairness, strong process fairness, U-fairness, and hyperfairness, and a possibility result for weak process fairness.

The impossibility results do not depend on the type of communication primitives (e.g., message-passing or shared-memory) provided by the underlying execution model. It holds as long as (1) one process's readiness for multiparty interaction can be known by another only through communication, and the time it takes two processes to communicate is nonnegligible (but can be finitely bounded); and (2) the time when a process will make its transition to this ready state (from a state not willing to engage in any interaction) cannot be determined a priori. Algorithms which claim any of these "impossible" fairness notions, therefore, must make use of some assumption which contradicts one of the two conditions, or assume a system topology which complies with the structure we have analyzed in the paper that can render the possibility phenomena (see Theorems 5.2 and 5.4 and Corollaries 5.9 and 5.12).

For example, Attie *et al.* [5] propose a distributed multiparty interaction scheduling algorithm fulfilling U-fairness. Their algorithm does not assume any system topology, and so is general for all interaction systems. However, they do implicitly assume that the time a process can stay in an idle state and the time it takes to execute an interaction are both finitely bounded. From time to time, a coordinating process has to pause its coordination activity, waiting for some process to be ready for an interaction even if there is another interaction enabled for execution. The delay imposed by the coordinator implies that the time it takes to schedule one interaction may depend on the other processes not involved in the interaction. From the efficiency's concern, this violates one of the four criteria proposed by Buckley and Silberschatz [14] for evaluating distributed interaction scheduling algorithms. Note that if the "bounded transition time" assumption is removed, the above algorithm would be deadlocked if the target process waited for by the coordinator is no longer interested in interaction.

Many algorithms for scheduling multiparty interactions that conform to the above two assumptions have also been proposed, e.g., [9, 21, 26, 29, 33, 42, 43, 45, 47]. From our results, it is not surprising to see that only few of them have claimed a fairness notion stronger than weak interaction fairness. (*Weak interaction fairness* requires an interaction that is continuously enabled to be established eventually, and so is much weaker than all fairness notions discussed in Section 5.) In particular, the algorithms of [26, 45, 47] also satisfy SPF with the proviso that interactions must be strictly bipartied, which, by our results in Section 5.2, is indeed possible to implement.

For other fairness notions that satisfy the criterion, we have also presented a general algorithm to implement them. The algorithm employs a centralized coordinator to simulate the behavior of the nonblocking scheduler characterized by the criterion. Our future work will focus on a distributed implementation for our criterion. That is, nonconflicting interactions can be established concurrently by different coordinators.

The impossibility results for SIF and SPF have also been established independently by Tsay and Bagrodia [46] and by Joung [26]. Our impossibility results for SIF and SPF improve upon theirs in

three ways: First, our results do not depend on any system topology underlying the implementation. By contrast, each process in [46] is paired with a coordinating process to schedule interactions, while [26] assumes a centralized coordinator for the scheduling. Second, they establish the impossibility results by identifying a particular system for which SIF and SPF are not possible. We are able to determine the structure of systems that renders the impossibility phenomena, and using the criterion we can also determine if SIF and SPF are implementable for any given specific system. Finally, and most importantly, they observe the impossibility phenomena in a specific implementation model. We, however, have generalized the model and lifted its properties to the semantic level. In effect, this reduces reasoning about a complex and concrete implementation model to reasoning about a simpler and abstract model for process interaction, and allows the criterion to apply to *every* possible fairness notion for multiparty interaction.

It should be noted that when we say that a fairness notion $\mathbb{C}$ is not implementable, we mean that there exists an interaction system for which $\mathbb{C}$ cannot be implemented by any deterministic algorithm. An unimplementable fairness notion may be implementable for some specific interaction system. The criterion we have proposed allows us to determine whether a fairness notion is implementable for any given interaction system. However, as we have also analyzed in the paper, the problem of determining whether an unimplementable fairness notion is implementable for some specific interaction system may turn out to be undecidable!

Furthermore, the multiparty interactions we have addressed in the paper assumed that the participants of an interaction are fixed in advance. This form of interactions has been widely used in distributed languages that support multiparty interactions. The participants of an interaction may also be parameterized, or even dynamically configured. In the latter case, determining the participants of an interaction could become very complex, and even intractable. A taxonomy of programming languages offering linguistic support for multiparty interaction along with a comprehensive complexity analysis of interaction membership decision problem is given by Joung and Smolka [30]. It is easy to see that the impossibility results we established in the paper also apply to other forms of multiparty interactions in which the participants of an interaction may vary dynamically.

Since deterministic algorithms are not possible for most fairness notions, *randomization* might be appealing. Randomization has proven to be an effective technique for coping with some impossibility phenomena occurring in the Dining Philosophers problem and CSP-like biparty interaction [20, 35, 44]. In fact, randomization is also effective for the more general problem, viz., the multiparty interaction scheduling. Joung and Smolka [31] present a symmetric, distributed, and randomized algorithm that, with probability 1, satisfies SIF. It thus offers an appealing tonic to other fairness notions lacking deterministic realizations.

ACKNOWLEDGMENTS

The author thank Reino Kurki-Suonio for his encouragement to carry out this research and his comments on an earlier version of this paper. The author is also deeply grateful to the anonymous referees for their thorough reading of the manuscript. Their numerous comments and suggestions have helped to significantly improve the paper.

REFERENCES

1. Abadi, M., and Lamport, L. (1991), The existence of refinement mappings, *Theoret. Comput. Sci.* **82**(2), 253–284.
2. Abadi, M., and Lamport, L. (1993), Composing specifications, *ACM Trans. Programming Lang. Syst.* **15**(1), 73–132.
3. Andrews, G. R., Olsson, R. A., Coffin, M., Elshoff, I., Nilsen, K., and Purdin, T. (1988), An overview of the SR language and implementation, *ACM Trans. Programming Lang. Syst.* **10**(1), 51–86.
4. Apt, K. R., Francez, N., and Katz, S. (1988), Appraising fairness in languages for distributed programming, *Distrib. Comput.* **2**(4), 226–241.
5. Attie, P. C., Forman, I. R., and Levy, E. (1990), On fairness as an abstraction for the design of distributed systems. *in* "Proceedings of the 10th International Conference on Distributed Computing Systems, Paris, France," pp. 150–157.
6. Attie, P. C., Francez, N., and Grumberg, O. (1993), Fairness and hyperfairness in multi-party interactions, *Distrib. Comput.* **6**, 245–254.
7. Back, R. J. R., and Kurki-Suonio, R. (1988), Distributed cooperation with action systems, *ACM Trans. Programming Lang. Syst.* **10**(4), 513–554.

8. Back, R. J. R., and Kurki-Suonio, R. (1988), Serializability in distributed systems with hand-shaking, *in* "Proceedings of the 15th International Colloquium on Automata, Languages and Programming, Tampere, Finland," Lecture Notes in Computer Science, Vol. 317, pp. 52–66, Springer-Verlag, Berlin.
9. Bagrodia, R. L. (1989), Process synchronization: Design and performance evaluation of distributed algorithms, *IEEE Trans. Software Eng.* **SE-15**(9), 1053–1065.
10. Bagrodia, R. L. (1989), Synchronization of asynchronous processes in CSP, *ACM Trans. Programming Lang. Syst.* **11**(4), 585–597.
11. Bolognesi, T., and Brinksma, E. (1987), Introduction to the ISO specification language LOTOS, *Comput. Networks ISDN Syst.* **14**, 25–59.
12. Bougé, L., and Francez, N. (1988), A compositional approach to superimposition, *in* "Proceedings of the 15th ACM Symposium on Principles of Programming Languages, San Diego, California," pp. 240–249, ACM Press, New York.
13. Brinksma, E. (1988), "On the Design of Extended LOTOS—A Specification Language for Open Distributed Systems," Ph.D. thesis, University of Twente, The Netherlands.
14. Buckley, G. N., and Silberschatz, A. (1983), An effective implementation for the generalized input–output construct of CSP, *ACM Trans. Programming Lang. Syst.* **5**(2), 223–235.
15. Charlesworth, A. (1987), The multiway rendezvous, *ACM Trans. Programming Lang. Syst.* **9**(2), 350–366.
16. Coffin, M., and Olsson, R. A. (1989), An SR approach to multiway rendezvous, *Comput. Lang.* **14**(4), 255–262.
17. Evangelist, M., Francez, N., and Katz, S. (1989), Multiparty interactions for interprocess communication and synchronization, *IEEE Trans. Software Eng.* **SE-15**(11), 1417–1426.
18. Francez, N., and Forman, I. R. (1996), "Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming," Addison–Wesley, Reading, MA.
19. Francez, N., Hailpern, B., and Taubenfeld, G. (1986), Script: A communication abstraction mechanism, *Sci. Comput. Programming* **6**(1), 35–88.
20. Francez, N., and Rodeh, M. (1980), A distributed abstract data type implemented by a probabilistic communication scheme, *in* "Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science, Long Beach, California," pp. 373–379.
21. Gao, Q., and Bochmann, G. V. (1989), Distributed implementation of LOTOS multi-rendezvous, *in* "Participants Proceedings of the 9th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification (E. Brinksma, G. Scollo, and C. Vissers, Eds.), University of Twente, The Netherlands.
22. Hoare, C. A. R. (1978), Communicating sequential processes, *Commun. ACM* **21**(8), 666–677.
23. Hoare, C. A. R. (1984), "Occam Programming Manual," Prentice–Hall, New York.
24. Järvinen, H.-M., and Kurki-Suonio, R. (1991), DisCo specification language: Marriage of actions and objects, *in* "Proceedings of the 11th International Conference on Distributed Computing Systems, Arlington, TX," pp. 142–151, IEEE Computer Society Press, New York.
25. Järvinen, H.-M., Kurki-Suonio, R., Sakkinen, M., and Systä, K. (1990), Object-oriented specification of reactive systems, *in* "Proceedings of the 12th International Conference on Software Engineering, Nice, France," pp. 63–71, IEEE Computer Society Press, NewYork.
26. Joung, Y.-J. (1992), "On the Design and Implementation of Multiparty Interaction," Ph.D. thesis, Department of Computer Science, State University of New York at Stony Brook.
27. Joung, Y.-J. (1996), Characterizing fairness implementability for multiparty interaction, *in* "Proceedings of the 23rd International Colloquium on Automata, Languages and Programming, Paderborn, Germany," pp. 110–121, Lecture Notes in Computer Science, Vol. 1099, Springer-Verlag, Berlin.
28. Joung, Y.-J. (2001), On fairness notions in distributed systems. II. Equivalence-completions and their hierarchies, *Inform. Comput.* **166**, 35–60.
29. Joung, Y.-J., and Smolka, S. A. (1994), Coordinating first-order multiparty interactions, *ACM Trans. Programming Lang. Syst.* **16**(3), 954–985.
30. Joung, Y.-J., and Smolka, S. A. (1996), A comprehensive study of the complexity of multiparty interaction, *J. ACM* **43**(1), 75–115.
31. Joung, Y.-J., and Smolka, S. A. (1998), Strong interaction fairness via randomization, *IEEE Trans. Parallel Distrib. Syst.* **9**(2), 137–149.
32. Katz, S., Forman, I., and Evangelist, M. (1990), Language constructs for distributed systems, *in* "IFIP TC2 Proceedings of Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel," pp. 70–97.
33. Kumar, D. (1990), An implementation of N-party synchronization using tokens, *in* "Proceedings of the 10th International Conference on Distributed Computing Systems, Paris," pp. 320–327.
34. Lamport, L. (1978), Time, clocks and the ordering of events in a distributed system, *Commun. ACM* **21**(7), 558–565.
35. Lehman, D., and Rabin, M. O. (1981), On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract), *in* "Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages," pp. 133–138, ACM Press, New York.
36. Milne, G. J. (1985), CIRCAL and the representation of communication, concurrency, and time, *ACM Trans. Programming Lang. Syst.* **7**(2), 270–289.
37. Milner, R. (1983), Calculi for synchrony and asynchrony, *Theoret. Comput. Sci.* **25**, 267–310.
38. Milner, R. (1989), "Communication and Concurrency," International Series in Computer Science. Prentice–Hall, London.
39. Milner, R., Parrow, J., and Walker, D. (1992), A calculus of mobile processes, I, *Inform. Comput.* **100**(1), 1–40.
40. Olderog, E. R., and Apt, K. R. (1988), Fairness in parallel programs: The transformational approach, *ACM Trans. Programming Lang. Syst.* **10**(3), 420–455.

41. Owicki, S., and Lamport, L. (1982), Proving liveness properties of concurrent programs, *ACM Trans. Programming Lang. Syst.* **4**(3), 455–495.
42. Park, M. H., and Kim, M. (1990), A distributed synchronization scheme for fair multi-process handshakes, *Inform. Process. Lett.* **34**, 131–138.
43. Ramesh, S. (1987), A new and efficient implementation of multiprocess synchronization, *in* "Proceedings, Conference on PARLE," Lecture Notes in Computer Science, Vol. 259, pp. 387–401, Springer-Verlag, Berlin.
44. Reif, J. H., and Spirakis, P. G. (1984), Real time synchronization of interprocess communications, *ACM Trans. Programming Lang. Syst.* **6**(2), 215–238.
45. Sistla, P. A. (1984), Distributed algorithms for ensuring fair interprocess communications, *in* "Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing," pp. 266–277, ACM Press, New York.
46. Tsay, Y.-K., and Bagrodia, R. L. (1993), Some impossibility results in interprocess synchronization, *Distrib. Comput.* **6**(4), 221–231.
47. Tsay, Y.-K., and Bagrodia, R. L. (1994), Fault-tolerant algorithms for fair interprocess synchronization, *IEEE Trans. Parallel Distrib. Syst.* **5**(7), 737–748.
48. U.S. Department of Defense (1983), "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A, U.S. Government Printing Office, Washington, DC.