Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

SOME IMPOSSIBIITY RESULTS IN INTERPROCESS SYNCHRONIZATION

Y.-K. Tsay
R. L. Bagrodia

May 1992
CSD-920023

Some Impossibility Results in Interprocess Synchronization[1]

Yih-Kuen Tsay
Rajive L. Bagrodia

3531 Boelter Hall
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024.
Tel: (213) 825-0956

yihkuen@cs.ucla.edu
bagrodia@cs.ucla.edu

Abstract. In this paper we construct a formal specification of the problem of synchronizing asynchronous processes under strong fairness. We prove that strong interaction fairness is impossible for binary (and hence for multiway) interactions and strong process fairness is impossible for multiway interactions.

[1]This research was partially supported by NSF under grant ASC9157610 and by ONR under grant N00014-91-J-1605. This paper is a revision of an earlier UCLA Technical Report CSD-890059 with the same title.

1 Introduction

The problem of synchronizing asynchronous processes in a distributed environment was introduced in the context of the rendezvous construct proposed for CSP [Hoa78]. A rendezvous represents synchronous communication between two processes in which the sender (receiver) process must wait until the receiver (sender) process is ready to receive (send) the message. A process may be ready to rendezvous with a number of other processes but may participate in at most one rendezvous at a time. This form of communication is referred to as a binary interaction. Other researchers have suggested an extension to binary interaction — multiway interaction [FHT86, Cha87, BKS88, Fra89]. A multiway interaction essentially allows a rendezvous to occur among an arbitrary (though usually predetermined) number of processes.

A large number of algorithms have been devised to implement binary and multiway interactions [BS83, Sis84, Ram87b, Bag89b, Ram87a, CM88, Bag89a]. Three primary classes of properties are associated with such algorithms: safety, liveness, and fairness. Although most existing algorithms satisfy the safety and liveness properties, few implement fairness.

In this paper we formally specify the problem of implementing interactions in a general model. We consider two fairness notions: Strong Interaction Fairness (SIF), which requires that if an interaction is enabled infinitely often it be started infinitely often and Strong Process Fairness (SPF), which requires that if a process is ready to participate in some enabled interaction infinitely often it do so infinitely often. We prove that, in general, SIF is impossible for binary (and hence for multiway) interactions and SPF is impossible for multiway interactions. Although the impossibility results are proven in a message-passing model of distributed system, the results hold in any model where (a) each process autonomously decides when and if it is willing to participate in some interaction and (b) the model assumes low atomicity, i.e. in one atomic step a process cannot both change its local state and inform other processes of the change.

In [Fra86] Francez gives an extensive overview of fairness notions and demonstrates the effects of some of them on program correctness. [AFK88] proposes criteria for determining the appropriateness of fairness notions in distributed languages. They conclude that, under the suggested criteria, none of the common forms of fairness (including SPF and SIF) are appropriate for multiway interactions and only SPF is appropriate for binary interactions. Our impossibility results corroborate their conclusions; in another paper [BT90], we give an efficient algorithm for binary interactions with SPF. From an implementation perspective, Dijkstra [Dij88] contends that fairness is a void obligation for language implementors in that it is impossible to detect if the obligation has been fulfilled. The results of this paper show that under the assumptions of our model, some fairness notions for interprocess synchronization are, in fact, impossible to implement.

The rest of the paper is organized as follows: Section 2 describes the interaction problem and informally presents the impossibility results. Sections 3–5 establish the necessary framework and give a formal proof of the impossibility results. These sections may be omitted by readers who are less interested in the proof procedure. The last section examines the assumptions in the interaction problem, their possible alternatives, and related issues.

2 Informal Approach

We informally define the interaction problem and sketch the impossibility results, focusing on the crucial assumptions of the problem and their consequence to the results. All technical terms (*italic*-typed in their first appearances in this section) will be made formal and precise in subsequent sections.

2.1 The Problem

Consider a set of *processes* and a set of *interactions* defined among the processes. Each interaction is a nonempty subset of processes representing some synchronization activity of its members. A process can be in *active* or *idle* state. An active process may *autonomously* become idle and wait to participate in some interaction. (Note that in general it is impossible to determine a priori when, or if, an active process will become idle.) An interaction is *enabled* if all of its members are idle; it is *disabled* otherwise. A process is said to be *committed* if some interaction of which it is a member is *started*; an idle process may become active only if it is committed. A started interaction will eventually be terminated.

It is required to *augment* each process with a *scheduler* to select interactions for execution such that the following safety and liveness properties are satisfied: (a) Only enabled interactions can be started.[1] (b) A process can participate in at most one interaction *at a time*. Two interactions are said to be *conflicting* if they have at least one member in common. This property says that once an interaction is started, no other conflicting interactions may be started until the interaction is terminated. (c) If an interaction is enabled, then either the interaction or some other conflicting interaction will eventually be started.

This problem is referred to as the multiway interaction problem; if each interaction has exactly two members, the problem is referred to as the binary interaction problem. The processes in an instance of the problem, together with their schedulers will be referred to as a *program*.

[1] An interaction must be enabled at the time when it is started. However, an interaction that has been started may subsequently become disabled when some of its members become active. A process that becomes idle while some interaction of which it is a member has been started is considered to be participating in the started interaction.

2.2 Fairness Notions

Aside from the basic properties required by the interaction problem, it is natural to ask whether stronger properties can be satisfied. The problem description in the preceding section allows a *run* of a program, where an interaction I is enabled infinitely many times but is never started because some member of I always chooses to participate in a conflicting interaction.

We thus consider two stronger properties: *Strong Interaction Fairness* (SIF) and *Strong Process Fairness* (SPF). SIF requires that if an interaction is enabled *infinitely often* it be started infinitely often; and SPF requires that if a process is ready to participate in some enabled interaction infinitely often it do so infinitely often. Notice that SIF subsumes SPF; a program that violates SPF cannot satisfy SIF.

2.3 Impossibility Results

We show that, in general, SIF is impossible for the binary interaction problem and hence for the multiway interaction problem. The impossibility of SPF for multiway interactions follows immediately from the preceding result.

In the description of the interaction problem, we have made three assumptions which are crucial to the impossibility results: (i) An active process may or may not become idle. (ii) If an active process becomes idle, it does so autonomously. (iii) The state transition of a process is not immediately observable by other processes or their schedulers. Modifications to the assumptions and their effect on the impossibility results are examined in Section 6.

Consider an instance of the binary interaction problem with three processes i, j, and k and three interactions $\{i,j\}$, $\{j,k\}$, and $\{k,i\}$. The schedulers are referred to as schedulers i, j, and k, respectively. In accordance with assumptions (i) and (ii), we further assume that the three processes have the following property: (iv) It is always possible for an active process to remain active or autonomously become idle. We shall construct a run of the program where interaction $\{k,i\}$ becomes enabled infinitely often but is never started.

The construction is developed around the following key observations: First, if some interaction I is enabled, the schedulers cannot indefinitely postpone the execution of I while waiting for other interactions to become enabled; otherwise, if other interactions never become enabled as allowed by assumption (iv), property (c) in Section 2.1 will be violated. This observation will be established formally by Lemma 3 in Section 5. Secondly, when the schedulers decide to start some interaction I, it is possible that a conflicting interaction K becomes enabled before I is started. However, assumption (iii) implies that in general it is impossible for the schedulers to detect that K is enabled before I is started. This will be established formally in Theorem 1 in Section 5.

In some run of the program, the following scenario may occur repeatedly: Initially, all processes are active and no interaction is started. Processes i and j go from active to idle, while process k remains active. To satisfy property (c), schedulers i and j decide to start interaction $\{i,j\}$. Meanwhile, process k becomes idle right before interaction $\{i,j\}$ is actually started thus causing interactions $\{j,k\}$ and $\{k,i\}$ to also become enabled together with interaction $\{i,j\}$. After participating in interaction $\{i,j\}$, process j becomes idle again causing $\{j,k\}$ to become enabled, while process i remains active. Schedulers j and k then start interaction $\{j,k\}$. Subsequently, processes j and k again become active and interaction $\{j,k\}$ is terminated. Now, all processes are active and no interaction is started; and the above scenario is repeated.

To see that SPF is impossible for multiway interactions, we add process l to the previous instance and change interaction $\{k,i\}$ into $\{k,i,l\}$. In a run of the new program, assume process l becomes idle. Processes i, j, and k behave exactly the same as above. It follows that process l is ready to participate in interaction $\{k,i,l\}$ infinitely often but never does so.

In the remainder of the paper, we define a computational model, construct a formal specification of the interaction problem with strong fairness, and derive the impossibility results.

3 Model and Definitions

3.1 Program and Computation

A *program* consists of a set of variables and a set of (state transition) rules. Each variable may assume values in some domain, a subset of which is specified as possible initial values of the variable. Every program includes an auxiliary variable called **label**, which may assume the name of a rule or an initial value *null*. The *state* of a program is the tuple of values assumed by the program variables, including **label**; an *initial* state is a state satisfying the specification of initial values of the program variables. Each *rule* is specified by a unique non-*null* tag, called its name, a predicate on program states, called its guard, and a sequence of assignment statements, called its body. A rule is *enabled* at a program state if the state satisfies its guard; otherwise it is *disabled*.

A *computation* (or *run*) of a program starts from any initial state and goes on forever. In each step of the computation, a rule is selected nondeterministically for execution and the value of **label** is updated with the name of the selected rule. If the selected rule is enabled, its body is executed; nothing else happens otherwise. The execution of a (enabled or disabled) rule results in a deterministic state transition of the program. Thus, each computation uniquely determines an infinite sequence of program states. To exclude computations where a continuously enabled rule is indefinitely ignored, we postulate a fair selection criterion: each rule of the program is selected

infinitely many times (regardless of whether or not the rule is enabled) in a computation.[2]

We introduce some notations:

s (or s', s_0, s_1, ... etc.) denotes a program state.

x_i denotes the i-th element of sequence x. We assume the elements of a sequence are numbered from 0.

x^i denotes the suffix of sequence x starting from the i-th element, i.e. $x_i x_{i+1} x_{i+2} \cdots$.

$\langle s, x \rangle$ denotes the sequence of states determined by the execution of sequence of rules x starting from state s.

(s,x) denotes the last state in $\langle s, x \rangle$, assuming x is finite.

xy denotes the concatenation of sequences x and y, assuming x is finite. From the definition of the computational model, it follows that $(s, xy) = ((s, x), y)$.

$x \leq y$ denotes that sequence x is a prefix of sequence y.

$Init_{\mathcal{D}}$ denotes the predicate that specifies the initial states of a program $\mathcal{D}$.

$Rule(\mathcal{D})$ denotes the set of rules of $\mathcal{D}$.

$Rule^*(\mathcal{D}) \equiv \{\alpha \mid \forall r, k : r \in Rule(\mathcal{D}) \wedge k \geq 0 :: (\exists i : i \geq k :: \alpha_i = r)\}$ is the set of all infinite sequences of rules of $\mathcal{D}$ such that each rule is selected infinitely many times.

$Pref(\mathcal{D}) \equiv \{x \mid \exists \alpha : \alpha \in Rule^*(\mathcal{D}) :: x \leq \alpha\}$ is the set of all prefixes of sequences in $Rule^*(\mathcal{D})$.

$Comp(\mathcal{D}) \equiv \{\sigma \mid \exists s, \alpha : (s \Rightarrow Init_{\mathcal{D}}) \wedge \alpha \in Rule^*(\mathcal{D}) :: \sigma = \langle s, \alpha \rangle\}$ is the set of all possible computations of $\mathcal{D}$.

$Comp^*(\mathcal{D}) \equiv \{\sigma \mid \exists x :: x\sigma \in Comp(\mathcal{D})\}$ is the suffix closure of $Comp(\mathcal{D})$.

$Branch(s) \equiv \{\sigma \mid \sigma \in Comp^*(\mathcal{D}) \wedge \sigma_0 = s\}$ is the set of all possible "futures" of the state s. Note that each possible future of a program state is a sequence of states.

3.2 Temporal Logic

Our logic is a variation of the branching time logic in [ES89]. We do not distinguish between state formulae and path formulae, but simply refer to them as temporal formulae. We use $\bigcirc, \Box, \Diamond$ instead of the more standard *X,G,F*. Quantifiers are introduced to abbreviate the conjunction or disjunction of a number of temporal formulae with similar pattern.

We directly define the semantics of our logical language with respect to a program $\mathcal{D}$; its syntax is implicitly defined by these semantic definitions. Suppose *a,b,c* are predicates on program states and *p,q* are temporal formulae. σ is an infinite sequence in $Comp^*(\mathcal{D})$; recall that σ^i denotes its suffix $\sigma_i \sigma_{i+1} \sigma_{i+2} \cdots$. In the following definitions, the logical operators $\neg$, $\wedge$, and $\vee$ and quantifiers

[2]The fair selection criterion only requires that each rule be selected infinitely often. In particular, it is possible that a rule is enabled infinitely often (not continuously) but the body of the rule is never executed, because it may be selected only when it is disabled.

$\forall$ and $\exists$, when not occurring as part of a temporal formula, should be interpreted according to their standard meanings in classical logic.

$$a \mid \sigma \equiv a \text{ at } \sigma_0 \quad (a \text{ is evaluated to } \textit{true} \text{ at state } \sigma_0) \tag{A1}$$

$$\neg p \mid \sigma \equiv \neg(p \mid \sigma) \tag{A2}$$

$$\bigcirc p \mid \sigma \equiv p \mid \sigma^1 \tag{A3}$$

$$\Box p \mid \sigma \equiv \forall i : i \geq 0 :: p \mid \sigma^i \tag{A4.1}$$

$$\Diamond p \mid \sigma \equiv \exists i : i \geq 0 :: p \mid \sigma^i \quad (= \neg\Box\neg p \mid \sigma) \tag{A4.2}$$

$$\mathsf{A} p \mid \sigma \equiv \forall \tau : \tau \in \mathit{Branch}(\sigma_0) :: p \mid \tau \tag{A5.1}$$

$$\mathsf{E} p \mid \sigma \equiv \exists \tau : \tau \in \mathit{Branch}(\sigma_0) :: p \mid \tau \quad (= \neg\mathsf{A}\neg p \mid \sigma) \tag{A5.2}$$

$$p \vee q \mid \sigma \equiv (p \mid \sigma) \vee (q \mid \sigma) \tag{A6.1}$$

$$p \wedge q \mid \sigma \equiv (p \mid \sigma) \wedge (q \mid \sigma) \quad (= \neg(\neg p \vee \neg q) \mid \sigma) \tag{A6.2}$$

$$p \Rightarrow q \mid \sigma \equiv (\neg p \vee q) \mid \sigma \tag{A6.3}$$

$$p \Leftrightarrow q \mid \sigma \equiv ((p \Rightarrow q) \wedge (q \Rightarrow p)) \mid \sigma \tag{A6.4}$$

$$(a \textit{ Until } b) \mid \sigma \equiv \exists i : i \geq 0 :: (b \mid \sigma^i) \wedge (\forall j : 0 \leq j < i :: a \mid \sigma^j) \tag{A7.1}$$

$$(a \textit{ Unless } b) \mid \sigma \equiv (a \Rightarrow (\Box a \vee (a \textit{ Until } b))) \mid \sigma \tag{A7.2}$$

(Notice that "$(a \textit{ Unless } b) \mid \sigma$" is true if a is is false at σ_0, regardless of the truth value of b. This definition, motivated by the "*unless*" in [CM88], is very useful in specifying safety properties of a program.)

A quantified temporal formula is interpreted as multiple occurrences of the temporal formula with the quantified variables replaced by their possible values. "$(\forall x : Q(x) :: p(x)) \mid \sigma$" is evaluated to true if all occurrences of $p(x)$ with x satisfying $Q(x)$ are evaluated to true. *An important constraint on the predicate $Q(x)$ is that its truth value does not depend on program states.* For example, x can be the index of processes and $Q(x)$ asserts that x range over some set of numbers. Similarly, "$(\exists x : Q(x) :: p(x)) \mid \sigma$" is evaluated to true if at least one occurrence of $p(x)$ with x satisfying $Q(x)$ is evaluated to true. For brevity, temporal formulae will often be written without explicit quantification; they are assumed to be universally quantified over all values of the free variables.

The properties of a program $\mathcal{D}$ are expressed by statements of the form "p in $\mathcal{D}$," where p is a temporal formula.

$$p \text{ in } \mathcal{D} \equiv \forall \sigma : \sigma \in \mathit{Comp}^*(\mathcal{D}) :: p \mid \sigma \quad (= \forall \tau : \tau \in \mathit{Comp}(\mathcal{D}) :: \Box p \mid \tau) \tag{P1}$$

The following are some temporal formulae that are true for any sequence of $\mathit{Comp}^*(\mathcal{D})$. Their validity can easily be verified from the definitions (A1)–(A7.2). Notice again that a, b, c, and d are predicates on program states and do not involve temporal operators.

$$\Box(p \wedge q) \Leftrightarrow (\Box p \wedge \Box q) \tag{T1}$$

$$(a \wedge \mathsf{A} p) \Leftrightarrow \mathsf{A}(a \wedge p) \tag{T2}$$

$$(\Box(p \Rightarrow q) \wedge \Box p) \Rightarrow \Box q \tag{T3}$$

$$(a \ \textit{Unless} \ (b \vee c)) \Rightarrow (a \ \textit{Unless} \ b) \vee (a \ \textit{Unless} \ c) \tag{T4}$$

$$(a \ \textit{Unless} \ b) \wedge (c \ \textit{Unless} \ d) \Rightarrow (a \wedge c \ \textit{Unless} \ b \vee d) \tag{T5}$$

3.3 Program Composition and Distributed Programs

Programs can be combined to produce composite programs in a natural way. Each component program of a composite program will be referred to as a *module*. The set of variables (rules) of the composite program is the union of the sets of variables (rules) of all modules. Variables belonging to more than one modules are termed *shared* variables. A constraint on program composition requires that each shared variable be initialized "consistently" by all sharing modules. A program composed of modules F and G is denoted by $F[\!]G$. Note that F and G may themselves be composite programs. In a computation of $F[\!]G$, each rule of F or G must be selected infinitely often. A computation of F is no longer a computation of $F[\!]G$, since the rules of G are not selected; analogously for G. For clarity, the state of a module in a composite program will be referred to as the *local* state of the module.

We consider programs where program modules are functionally divided into two categories: *processes* which do significant computations and *channels* which simply relay messages. Distinct processes have disjoint sets of variables and so do distinct channels; variables may be shared only between a process and a channel. A sender process may send a message to a receiver process by depositing the message in a message queue shared by the sender and a channel; the channel then delivers the message by removing the message and depositing it in another message queue shared by the channel and the receiver process. (Note that the notions of process and channel are relative to a program. A module in a process, which shares variables with other modules in the same process, is *not* a process of the entire program; analogously for modules in a channel.)

Programs composed in the above manner are called *distributed* programs. A distributed program models a distributed system with message passing.

4 Multiway and Binary Interaction Problems

Let *USER* refer to a distributed program which contains a set of asynchronous processes and the channels that relay messages among the processes and *OS* refer to the distributed scheduler that implements synchronizations among the asynchronous processes in *USER*. The composite program *USER*$[\!]$*OS* is referred to as $\mathcal{P}$.

A process in *USER* with index i is denoted by $\mathtt{user}_i$; analogously for *OS*. Let p_i denote $\mathtt{user}_i[\!]\mathtt{os}_i$; each p_i is a process in $\mathcal{P}$. We shall refer to a process in *USER* as a **user**, a process in

OS as an **os**, and a process in $\mathcal{P}$ as a **process**. An interaction among $\mathbf{user}_i$, $\mathbf{user}_j$, and $\mathbf{user}_k$ is represented by $\{i,j,k\}$. $\mathcal{I}$ is the set of all interactions defined among **users**; each element of $\mathcal{I}$ is a nonempty subset of the process indexes. Two interactions are said to be *conflicting* if they have at least one common member. The set of all interactions of which a **user** (or loosely, a **process**) is a member is referred to as the *interaction set* of the **user** (or **process**).

Each **user** and the corresponding **os** share two variables: **state** and **flag**. **state** may assume the value *active* or *idle*. The two states of a **user** (or **process**) correspond to a **user** that does not want to participate in any interaction and a **user** that is waiting to participate in some interaction. Interaction I is started if one of its members, say p_i, sets $\mathbf{flag}_i$ to I and is terminated if **flag** is set back to *null* for all members of I. We say a process is *committed* if some interaction in its interaction set is started. The relationship between a **user** and its **os** and that between two processes (each formed by the composition of a **user** and its **os**) are depicted in Figure 1. As a pair of processes in $\mathcal{P}$ communicate via channels and do not share variables, an update of $\mathbf{state}_i$ in any computation step is not observed by any p_j $(j \neq i)$ in the next step, enforcing assumption (iii) in Section 2.3.

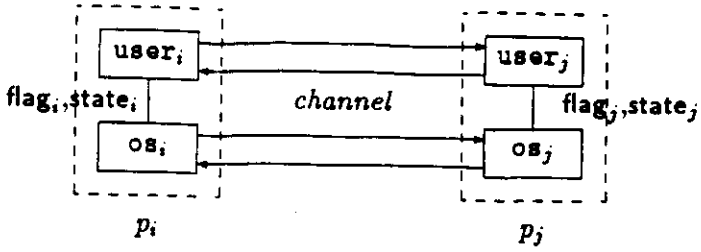


Figure 1: Compositions of **users** and **os**'s

We introduce some abbreviations for commonly used predicates:

$active_i \equiv (\mathbf{state}_i = active)$, analogously for $idle_i$ (d1)

$enable^I \equiv (\forall i : i \in I :: idle_i)$ (d2)

$start^I \equiv (\exists i : i \in I :: \mathbf{flag}_i = I)$ (d3)

$commit_i \equiv (\exists I : i \in I :: start^I)$ (d4)

$E[I,J] \equiv (I \neq J \wedge I \cap J \neq \phi)$, interactions I and J are conflicting (d5)

We use the temporal logic language introduced in Section 3.2 to specify the properties of *USER* and $\mathcal{P}$ as well as the constraints on *OS*.[3] Again, all temporal formulae are assumed to be universally quantified over all values of their free variables.

[3] We omit the exact temporal specification of the initial states as well as the precise specification of other restrictions (e.g. the restriction on how variables are shared among different modules). Whereas, the omitted specifications can easily be verified from the program text. Their explicit inclusion would considerably lengthen the problem specification and the impossibility proofs.

4.1 Specification of *USER*

This part specifies the behavior of the *USER* program at its interface with the *OS* and also specifies some properties that are guaranteed when *USER* is composed with the *OS*.

For each **user**, the variable **state** is initialized to *active* and **flag** to *null*. A **user** may not start an interaction — (u1). Provided that an **os** may not terminate an interaction and an **os** may not change the state of a **user** ((o1), (o2.1), and (o2.2) in Section 4.3), the *USER* will satisfy the following two properties: An idle **user** may become active only after some interaction in its interaction set is started — (u2) and a started interaction will eventually be terminated — (u3).

$(\mathbf{flag}_i = \mathit{null}) \wedge \bigcirc(\mathbf{label} \in \mathit{Rule}(\mathit{USER})) \Rightarrow \bigcirc(\mathbf{flag}_i = \mathit{null})$ in $\mathcal{P}$ (u1)

If *OS* satisfies (o1), (o2.1), and (o2.2) in Section 4.3, then

idle_i *Unless* commit_i in $\mathcal{P}$ (u2)

$\mathit{start}^I \Rightarrow \Diamond\neg\mathit{start}^I$ in $\mathcal{P}$ (u3)

4.2 Specification of $\mathcal{P}$

This part specifies the safety and liveness properties that must be provided by the composition of *USER* and *OS*.

The safety properties require that only *enabled* interactions can be started — (pp1) and that conflicting interactions cannot be started simultaneously — (pp2). The liveness property requires that if an interaction I is *enabled*, either I or a conflicting interaction be eventually started — (pp3).

$\neg\mathit{start}^I$ *Unless* enable^I in $\mathcal{P}$ (pp1)

$E[I, J] \Rightarrow \neg(\mathit{start}^I \wedge \mathit{start}^J)$ in $\mathcal{P}$ (pp2)

$\mathit{enable}^I \Rightarrow \Diamond(\mathit{start}^I \vee (\exists J : E[I, J] :: \mathit{start}^J))$ in $\mathcal{P}$ (pp3)

4.3 Constraints on *OS*

The only shared variables between $\mathbf{user}_i$ and $\mathbf{os}_i$ are $\mathbf{state}_i$ and $\mathbf{flag}_i$. For each **os**, **state** is initialized to *active* and **flag** to *null* (consistent with the initialization in *USER*). An **os** may not terminate an interaction — (o1) and an **os** may not change the state of a **user** — (o2.1) and (o2.2).

$(\mathbf{flag}_i = I) \wedge \bigcirc(\mathbf{label} \in \mathit{Rule}(\mathit{OS})) \Rightarrow \bigcirc(\mathbf{flag}_i = I)$ in $\mathcal{P}$ (o1)

$\mathit{active}_i \wedge \bigcirc(\mathbf{label} \in \mathit{Rule}(\mathit{OS})) \Rightarrow \bigcirc\mathit{active}_i$ in $\mathcal{P}$ (o2.1)

$\mathit{idle}_i \wedge \bigcirc(\mathbf{label} \in \mathit{Rule}(\mathit{OS})) \Rightarrow \bigcirc\mathit{idle}_i$ in $\mathcal{P}$ (o2.2)

(The assumptions (i) and (ii) in Section 2.3 are enforced by (o2.1) and (o2.2).)

4.4 Remark

The specification in Section 4.1–4.3 is intended to be a general abstraction of the problem of implementing nondeterministic synchronous communications among asynchronous processes in a distributed system. In particular, no restriction regarding the number of processes or what interactions are defined is given. Possible state transitions of a process are also omitted; for example, a process that never becomes idle and a process that will always eventually become idle will both satisfy the specification.

A number of algorithms for the interaction problem as specified have been suggested [CM88, Bag89a]. This indicates that the specification is "consistent," or more precisely, for any *USER* (satisfying the specification of *USER*) there exists an *OS* (satisfying the constraints on *OS*) such that their composition satisfies the specification of $\mathcal{P}$.

4.5 Additional Properties: Fairness

We formally specify the two fairness properties introduced in Sections 1 and 2: SIF and SPF.

SIF $\equiv\ \Box\Diamond enable^I \Rightarrow \Box\Diamond start^I$

SPF $\equiv\ \Box\Diamond ready_i \Rightarrow \Box\Diamond(\exists I : i \in I :: start^I)$, where $ready_i\ \equiv\ (\exists J : i \in J :: enable^J)$

It can be shown that "SPF in $\mathcal{P}$", or simply as SPF, subsumes (pp3).

5 Impossibility Results

Given an additional property ϕ, a *USER* is said to be *ϕ-compatible* if there exists an *OS* such that their composition satisfies the specification of $\mathcal{P}$ and also the additional property ϕ; otherwise the *USER* is *ϕ-incompatible*. We prove that a *USER* is *ϕ-incompatible* by showing that, for any *OS* such that the composition of *USER* and *OS* satisfies the specification of $\mathcal{P}$, $Comp^*(\mathcal{P})$ always contains some sequence violating ϕ. We shall use this approach to prove that there are SIF-*incompatible* instances for the binary interaction problem and there are SPF-*incompatible* instances for the multiway interaction problem. As a consequence, SIF is in general impossible for binary or multiway interactions and SPF is in general impossible for multiway interactions.

We start with some general properties of distributed programs.

5.1 Characteristics of Distributed Programs

Consider a distributed program $\mathcal{D}$. Let Q be the composition of some modules in $\mathcal{D}$ and $\overline{Q}$ be the composition of some other modules such that Q and $\overline{Q}$ do not share any variables. $s[Q]$ denotes the projection of program state s on Q, i.e. the local state of Q at s. The following two lemmas describe conditions under which the projections of (possibly different) states of $\mathcal{D}$ on Q are equivalent.

These results capture the ideas behind fusion of computations in [CM86], which is one of the basic techniques in our impossibility proofs and in others, e.g. [FLP85].

Lemma 1 *If the local states of Q corresponding to two program states s and s' are the same, the execution of a sequence of rules in Q starting respectively from s and s' will also result in identical local states of Q. In other words, if $(s[Q] = s'[Q]) \wedge x \in Pref(Q)$, then $(s,x)[Q] = (s',x)[Q]$.*

Proof. According to our model, the execution of a rule of a program results in a deterministic state transition of the program. Starting from the same state, a program will reach a unique state after the execution of the same sequence of rules. Since Q is also a program, the lemma follows.

End of Proof.

Lemma 2 *The execution of a sequence of rules in $\overline{Q}$ has no effect on the local state of Q. In other words, if $(s[Q] = s'[Q]) \wedge x \in Pref(\overline{Q})$, then $s[Q] = (s',x)[Q]$.*

Proof. From the assumption, Q and $\overline{Q}$ do not share any variables. Also, by the definition of program, rules in $\overline{Q}$ may only reference variables in $\overline{Q}$ and cannot change the value of any variable in Q. The lemma follows immediately. *End of Proof.*

Lemma 2 has the following application: According to the problem specification, any pair of processes in $\mathcal{P}$ communicate via channels and do not share variables. The execution of a rule or a sequence of rules in p_i will not change the values of the variables in any p_j $(j \neq i)$; an update of $\mathbf{state}_i$ in any step is not observed by any p_j $(j \neq i)$ in the immediately following step. In general, the preceding applies to the composition of some **users** and **os**'s and the composition of some other **users** and **os**'s, as these two compositions do not share any variables.

5.2 Impossibility Proofs

In the following proofs, we consider $\mathcal{BIN}$, a collection of instances of the binary interaction problem in which *USER* has three processes $\mathbf{user}_i$, $\mathbf{user}_j$, and $\mathbf{user}_k$ and $\mathcal{I} = \{I,J,K\}$, where $I = \{i,j\}$, $J = \{j,k\}$, and $K = \{k,i\}$.

Besides (u1)-(u3), the three processes also satisfy the following properties: A **user** becomes idle due to the execution of a sequence of rules belonging to the same **user** — (u4) and, if the *OS* satisfies its constraints, it is always possible that an active **user** never becomes idle — (u5).

$active_i \Rightarrow \mathsf{E}\Diamond idle_i$ in $\mathbf{user}_i$, analogously for $\mathbf{user}_j$ and $\mathbf{user}_k$. (u4)

If *OS* satisfies (o1), (o2.1), and (o2.2), then

$active_i \Rightarrow \mathsf{E}\Box active_i$ in $\mathcal{P}$, analogously for $\mathbf{user}_j$ and $\mathbf{user}_k$. (u5)

The following lemma formally establishes the first key observation in Section 2.3, i.e. if some interaction I is enabled, the schedulers cannot indefinitely postpone the execution of I while waiting for other interactions to become enabled.

Lemma 3 *For any instance in $\mathcal{BIN}$, if interaction I is enabled, p_k is active (so, interactions J and K are disabled), and no interaction is started at some state, then there exists a possible future of the state in which interaction I is started and p_k remains active until I is started. In other words:*

$$(enable^I \wedge active_k \wedge \neg start^+) \Rightarrow \mathsf{E}((enable^I \wedge active_k \wedge \neg start^+ \ \mathit{Unless}\ start^I) \wedge \Diamond start^I) \text{ in } \mathcal{P},$$

where $start^+$ denotes $(start^I \vee start^J \vee start^K)$.

Proof. Assuming the contrary, we shall demonstrate a possible future of a state satisfying $(enable^I \wedge active_k \wedge \neg start^+)$ such that interaction I remains enabled but neither I, J, nor K will be started, violating (pp3).

$\exists \sigma : \sigma \in Comp^*(\mathcal{P}) :: (enable^I \wedge active_k \wedge \neg start^+) \wedge$
$\mathsf{A}\neg((enable^I \wedge active_k \wedge \neg start^+ \ \mathit{Unless}\ start^I) \wedge \Diamond start^I) \mid \sigma$
, from the contrary assumption and definitions (P1), (A5.2), and (A6.3).

Fix the above σ.

$\forall \tau : \tau \in Branch(\sigma_0) :: (enable^I \wedge active_k \wedge \neg start^+) \wedge$
$\neg((enable^I \wedge active_k \wedge \neg start^+ \ \mathit{Unless}\ start^I) \wedge \Diamond start^I) \mid \tau$
, from the above, (T2), and (A5.1). (1)

$\forall \tau : \tau \in Branch(\sigma_0) :: enable^I \wedge active_k \wedge \neg start^+ \mid \tau$
, from the above. (2)

$\forall \tau : \tau \in Branch(\sigma_0) :: \neg(enable^I \wedge active_k \wedge \neg start^+ \ \mathit{Unless}\ start^I) \vee \Box\neg start^I \mid \tau$
, from (1), (A4.2), and (A6.2). (3)

We deviate to prove the following property:

$(enable^I \wedge active_k \wedge \neg start^+ \ \mathit{Unless}\ start^I) \vee$
$(enable^I \wedge active_k \wedge \neg start^+ \ \mathit{Unless}\ start^J \vee start^K \vee idle_k)$ in $\mathcal{P}$. (4)

$\neg start^I \ \mathit{Unless}\ start^I$ in $\mathcal{P}$, analogously for J and K.
, from that an interaction is either started or not started.

$\neg start^+ \ \mathit{Unless}\ start^+$ in $\mathcal{P}$.
, (T5) on the above.

$active_k \ \mathit{Unless}\ idle_k$ in $\mathcal{P}$.
, from that a process is either *active* or *idle*.

$enable^I \ \mathit{Unless}\ start^+$ in $\mathcal{P}$.
, (T5) on (u2).

$enable^I \wedge active_k \wedge \neg start^+$ *Unless* $start^+ \vee idle_k$ in $\mathcal{P}$.

, (T5) on the above three.

Applying (T4) to the above, we obtain (4).

$\forall \tau : \tau \in Branch(\sigma_0) :: (enable^I \wedge active_k \wedge \neg start^+$ *Unless* $start^J \vee start^K \vee idle_k) \vee \Box \neg start^I \mid \tau$

, from (3) and (4). (5)

$\exists \tau' : \tau' \in Branch(\sigma_0) :: \Box active_k \mid \tau'$, or $\Box \neg idle_k \mid \tau'$

, from (u5) and (A5.2). (6)

Fix the above τ'.

$\neg start^J \wedge \neg start^K \wedge \Box(\neg enable^J \wedge \neg enable^K) \mid \tau'$

, from the above, (d2), and (2).

$\Box(\neg start^J \wedge \neg start^K) \mid \tau'$

, from the above, (pp1), and (A7.2).

$\Box(\neg start^J \wedge \neg start^K \wedge \neg idle_k) \mid \tau'$

, from the above and (6).

$enable^I \wedge \Box(\neg start^I \wedge \neg start^J \wedge \neg start^K) \mid \tau'$.

, from the above, (5), (A7.2), and (2).

The above violates (pp3). *End of Proof.*

Theorem 1 *USERs in $\mathcal{BIN}$ are* SIF*-incompatible. (So, in general,* SIF *is impossible for the binary or multiway interaction problem.)*

Proof. Starting from a state of $\mathcal{P}$ where all processes are active and no interaction is started (initial states are such states), we are able to construct an infinite sequence of rules α satisfying the fair selection criterion, i.e. $\alpha \in Rule^*(\mathcal{P})$, such that interaction K is enabled infinitely often but never started. Formally, $\exists \sigma : \sigma \in Comp^*(\mathcal{P}) :: (\exists I : I \in \mathcal{I} :: \Box\Diamond enable^I \wedge \Box \neg start^I \mid \sigma)$. The construction proceeds in phases, where each phase consists of four stages. During each phase all interactions are enabled at least once but only I and J are started. At the end of each phase the program reaches a state where all processes again become active and no interaction is started. To satisfy the fair selection criterion, each rule in $\mathcal{P}$ is selected at least once in each phase. Figure 2 outlines the major state transitions in various stages of a phase.

Starting from a state s_0 where all processes are active and no interaction is started, each phase proceeds as follows:

Stage 1: $\mathbf{user}_i$ and $\mathbf{user}_j$ become idle (so, interaction I is enabled), while $\mathbf{user}_k$ remains active.

Apply (u4) first to $\mathbf{user}_i$ then to $\mathbf{user}_j$ (or the other way around) to obtain a sequence x_1 consisting of rules of $\mathbf{user}_i$ and $\mathbf{user}_j$ such that $(idle_i \wedge idle_j)$ at (s_0, x_1). From (d2), $enable^I$ at (s_0, x_1). Those rules of $\mathbf{user}_i$ and $\mathbf{user}_j$ not selected can be arranged in arbitrary order to form a

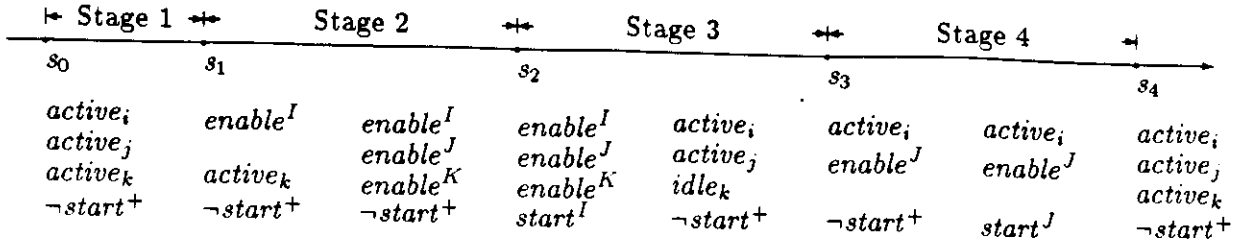


Figure 2: Major state transitions in a phase

sequence x_2. Let $s_1 = (s_0, x_1x_2)$. As x_1x_2 contains rules from only $\mathbf{user}_i$ and $\mathbf{user}_j$, due to (u1) and (u2), no interaction is started hence $enable^I \wedge \neg start^+$ at s_1; and, since $active_k$ at s_0 and x_1x_2 does not contain any rules in $\mathbf{user}_k$, from Lemma 2, we get $active_k$ at s_1.

Stage 2: Interaction I is started; $\mathbf{user}_k$ becomes idle just before I is started such that I, J, and K are enabled simultaneously. However, it is impossible for the schedulers to determine that J and K are enabled before I is started (the second key observation in Section 2.3).

Given $(enable^I \wedge active_k \wedge \neg start^+)$ at s_1, from Lemma 3, there exists a sequence y_1 of rules in $\mathcal{P}$ such that $start^I$ at (s_1, y_1) and $\neg start^J \wedge \neg start^K$ at all states in $\langle s_1, y_1 \rangle$. Without loss of generality, we assume $y_1 = y_0 r$, where r is a rule of $\mathbf{os}_i$ or $\mathbf{os}_j$, and $\neg start^I$ at (s_1, y_0), i.e. the execution of r starts interaction I. From (pp1) and $\neg start^I$ at (s_1, y_0), $enable^I$ at (s_1, y_0). According to (u4), there exists a sequence y_2 of rules in $\mathbf{user}_k$ such that $idle_k$ at (s_1, y_0y_2). (If $idle_k$ at some state in $\langle s_1, y_1 \rangle$, then y_2 is simply the empty sequence.) Let $s_1' = (s_1, y_0y_2)$. So, $(enable^I \wedge enable^J \wedge enable^K)$ at s_1'. Rule r can be selected for execution at s_1'.

From Lemma 2 (replacing s and s' in the lemma by (s_1, y_0), Q by $\mathbf{os}_i[\!]\mathbf{os}_j$, $\overline{Q}$ by $\mathbf{os}_k$, and x by y_2), $(s_1, y_0)[\mathbf{os}_i[\!]\mathbf{os}_j] = ((s_1, y_0), y_2))[\mathbf{os}_i[\!]\mathbf{os}_j]$, i.e. $(s_1, y_0)[\mathbf{os}_i[\!]\mathbf{os}_j] = s_1'[\mathbf{os}_i[\!]\mathbf{os}_j]$, which is to say that the transition to idle of $\mathbf{user}_k$ did not change the local state of $\mathbf{os}_i[\!]\mathbf{os}_j$ at (s_1, y_0).

From Lemma 1 (replacing s by (s_1, y_0), s' by s_1', Q by $\mathbf{os}_i[\!]\mathbf{os}_j$, and x by r),

$((s_1, y_0), r)[\mathbf{os}_i[\!]\mathbf{os}_j] = (s_1', r)[\mathbf{os}_i[\!]\mathbf{os}_j]$. As $y_1 = y_0 r$, we get $(s_1, y_1)[\mathbf{os}_i[\!]\mathbf{os}_j] = (s_1', r)[\mathbf{os}_i[\!]\mathbf{os}_j]$.

Since $start^I$ at (s_1, y_1), the preceding statement implies $start^I$ at (s_1', r). (1)

Also, from Lemma 2, $idle_k$ at s_1' implies $idle_k$ at (s_1', r). (2)

The above scenario is depicted in Figure 3, which shows that I is started irrespective of whether $\mathbf{user}_k$ is active or idle.

Those rules of $\mathcal{P}$ except $\mathbf{user}_i$ and $\mathbf{user}_j$, not selected in the sequence y_0y_2r can form an arbitrary sequence y_3. As y_3 does not contain any rules of $\mathbf{user}_i$ or $\mathbf{user}_j$, from Lemma 2 and (1), $start^I$ at all states in $\langle s_1', ry_3 \rangle$, which implies $\neg start^K$ at all states in $\langle s_1', ry_3 \rangle$, due to (pp2). According to (o2.2), (u2), and (2), $idle_k$ at (s_1', ry_3). Let $s_2 = (s_1', ry_3)$. So,

$(start^I \wedge idle_k)$ at s_2. (3)

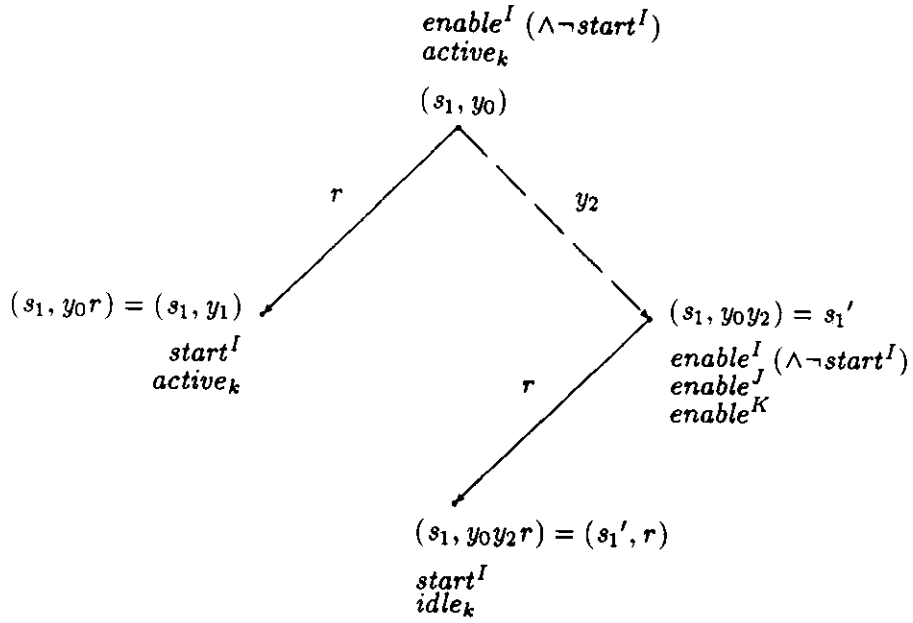


Figure 3: The transition to idle of $\mathbf{user}_k$ (execution of y_2) has no effect on starting interaction I (by the execution of r), as the local state transition of $\mathbf{user}_k$ is not immediately visible to $\mathbf{os}_i[\![\mathbf{os}_j$.

Stage 3: $\mathbf{user}_i$ becomes active and $\mathbf{user}_j$ becomes idle after a number of state transitions. Consequently, interaction J is enabled.

Interaction I was started in stage 2 at (s_1', r). By virtue of (u2), $\mathbf{user}_i$ and $\mathbf{user}_j$ go from *idle* to *active*; by (u4), interaction I is terminated. Similar to Stage 1, apply (u0.1) to $\mathbf{user}_j$ such that $\mathbf{user}_j$ becomes idle again; however $\mathbf{user}_i$ remains active. Let z be the corresponding sequence and $s_3 = (s_2, z)$. $idle_j$ at s_3 and $active_i$ at s_3. From (3) and Lemma 2, $idle_k$ at s_3. As z does not involve rules in $\mathbf{os}$'s, no more interaction is started. In summary, $(enable^J \wedge active_i \neg start^+)$ at s_3.

Stage 4: Similar to Stage 2, interaction J is started and, similar to Stage 3, both $\mathbf{user}_j$ and $\mathbf{user}_k$ eventually become active. Let w be the sequence and $s_4 = (s_3, w)$. All processes are active and no interaction is started at s_4.

All interactions are enabled in Stage 2 and interaction J is enabled in Stage 3. Interaction I is started in stage 2 and interaction J is started in Stage 4; while interaction K is never started. Repeat the four stages indefinitely, we obtain an infinite sequence α such that $(\Box\Diamond enable^K \wedge \Box\neg start^K) \mid \langle s_0, \alpha\rangle$. Each rule in $\mathcal{P}$ is selected at least once either in Stage 1 or Stage 2, so $\alpha \in Rule^*(\mathcal{P})$ and $\langle s_0, \alpha\rangle \in Comp^*(\mathcal{P})$. *End of Proof.*

Add one process $\mathbf{user}_l$, which has properties (u1)–(u5), to each *USER* in $\mathcal{BIN}$ and change K to $\{k,i,l\}$. We obtain a collection $\mathcal{MUL}$ of instances of the multiway interaction problem.

Theorem 2 *USERs in $\mathcal{MUL}$ are* SPF*-incompatible. (So, in general,* SPF *is impossible for the*

multiway interaction problem.)

Proof. At some point of computation, assume that p_l becomes idle, while other processes remain active. Since p_l may participate only in interaction K, in order to satisfy SPF, interaction K should be started infinitely often if it is enabled infinitely often. Ignore p_l altogether and treat this problem as implementing SIF for the equivalent binary interaction problem. The conclusion follows from Theorem 1.

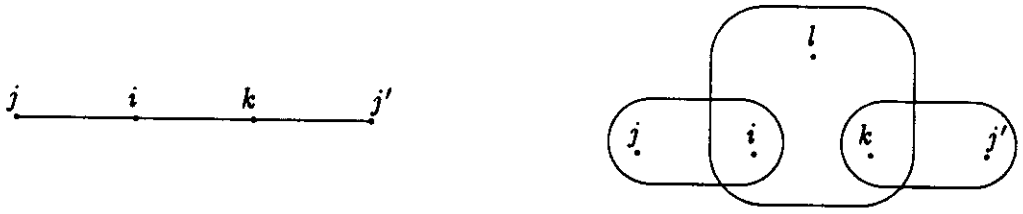
End of Proof.

6 Discussion

6.1 Identifying SIF/SPF-*incompatible* Problem Instances

Two factors determine whether a problem instance is SIF and/or SPF-*incompatible*: (a) the number of processes and the interactions defined among the processes and (b) the properties exhibited by the users in addition to (u1)–(u3).

For a given instance of the binary (or multiway) interaction problem, we view the set of processes and the set of interactions as an undirected graph (or hypergraph), called its *interaction graph*. A node in the interaction graph represents a process and an edge represents an interaction among the processes represented by the nodes incident to the edge. Problem instances in $\mathcal{BIN}$ and $\mathcal{MUL}$ have interaction graphs as shown in Figures 4 (a) and (b) respectively, where we identify j' with j.



(a) Binary (b) Multiway

Figure 4: "Minimal" Interaction Graphs, where j and j' may be identical

Assume that j and j' in Figures 4 (a) and (b) are distinct processes and that the interaction graphs represent problem instances in $\mathcal{BIN}'$ and $\mathcal{MUL}'$, respectively. Also assume that all users satisfy (u1)–(u5). Let J' denote interaction $\{j',k\}$. Lemma 3 and Theorems 1 and 2 still hold with $\mathcal{BIN}'$, $\mathcal{MUL}'$, and J' replacing $\mathcal{BIN}$, $\mathcal{MUL}$, and J, respectively. In fact, not every `user` needs to satisfy (u5). The proofs in Section 5.2 still work if only $\texttt{user}_i$ and $\texttt{user}_k$ are assumed to satisfy (u5). The interaction graph in Figure 4 (a) is "minimal" in the sense that an instance of the problem with a simpler interaction graph (with fewer processes or interactions) cannot be SIF-*incompatible*; analogously for Figure 4 (b).

There is a straightforward generalization of the preceding observation: Any problem instance with interaction graph reducible to Figure 4 (a) is SIF-*incompatible* and one reducible to Figure 4 (b) is SPF-*incompatible* provided that at least $\mathbf{user}_i$ and $\mathbf{user}_k$ satisfy (u5). (A graph is said to be *reducible* to another graph if the former graph is a result of removing some of the nodes or edges from the latter graph.)

We consider the impact of weakening **user** properties on the impossibility results: Assume that either $\mathbf{user}_i$ or $\mathbf{user}_k$ satisfies (u5) but which of them does is not known. Lemma 3 still holds and, as a consequence, Theorems 1 and 2 will also hold. However, the proof of Lemma 3 would involve game-playing arguments that assume an adversary rather than straightforward arguments based on the computational model and the temporal logic. It is also possible to assume a weaker property than (u5) for $\mathbf{user}_i$ and $\mathbf{user}_k$; in particular, it may be the case that a **user** is guaranteed to make only a finite number of transitions from active to idle. We believe that Theorems 1 and 2 are still true, though Lemma 3 is no longer true and the proofs of the theorems are no longer valid. However, the temporal formulation used in this paper is inadequate for developing a precise specification of the weaker assumption as well as the required proofs.

6.2 Basic Assumptions Revisited

We restate the basic assumptions of the problem and show that they are motivated by practical considerations. Other related issues are also discussed.

Assumptions:

(a) It is impossible to determine *a priori* whether an arbitrary process will make a transition from active to idle.

(b) The scheduler cannot control the actions of an active process. In particular, the scheduler cannot control when an active process becomes idle.

(c) The transition from *active* to *idle* of a process is not immediately observable by other processes or their schedulers.

The validity of assumption (a) can be proven by arguments similar to those used to demonstrate the undecidibility of the halting problem. For a set of processes with restricted behavior wherein it is possible to assume that every active process will eventually become idle, SIF can easily be guaranteed. Assume that some total order is assigned to the set of interactions. Given that at any point of a computation each active process will eventually become idle, every interaction must eventually become enabled. A scheduler may then simply choose each interaction in turn and wait until it is enabled (note that conflicting interactions that are simultaneously enabled will not be

executed, thus implying that the complexity of such algorithms be determined by the average time each process remains in the active state).

Contrary to assumption (b), it is possible to assume a more powerful scheduler which is responsible for scheduling both local and communication actions of a process. This would imply that the transition of a process from active to idle can also be controlled by the scheduler and indirectly by other processes in the system, thus violating the autonomy of a process in executing a local action. Such a scheduler can prevent a process from executing its active to idle transition, thus allowing it to control which interactions are enabled. In the extreme case, such a scheduler can always ensure that conflicting interactions are never enabled simultaneously and thus guarantee SIF in a straightforward manner. However, such a powerful scheduler is just an artifact that defines away the real problem. Furthermore, postulating such a powerful scheduler, in effect, implies that a scheduler has an instantaneous "global snapshot" about which interactions are enabled, a requirement that is met by very few real-life systems.

Assumption (c) is a consequence of unbounded communication delays in asynchronous systems. Moreover, SIF is impossible even if a known upper bound (other than zero) is assumed for communication delays. If the active to idle transition of a process is immediately observable by other processes in the system, SIF can be guaranteed, as once again, the scheduler has an instantaneous global snapshot of the enabled interactions.

Under the three assumptions, the impossibility results hold even in a model which allows more than one atomic actions (or rules) to be executed in each computation step (i.e. an overlapping model) in contrast to the interleaving model assumed in this paper. Also, it should be clear that any fairness property stronger than SPF will be impossible for multiway interactions and any fairness property stronger than SIF will be impossible for binary interactions. For example, a fairness property, which requires that two interactions which are enabled equally many times be started equally many times, is impossible for binary (and hence multiway) interactions. Interested readers are referred to [Fra86, AFK88] for stronger variations of SIF and SPF.

Acknowledgment

We are grateful to the referees for their detailed comments on earlier versions of the paper, which improved the paper significantly.

References

[AFK88] K. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[Bag89a] R.L. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, pages 1053–1065, September 1989.

[Bag89b] R.L. Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Languages and Systems*, 11(4):585–597, October 1989.

[BKS88] R.J. Back and R. Kurki-Suonio. Distributed cooperation with action sytstems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[BS83] G. Buckley and A. Silberschatz. An effective implementation of the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[BT90] R.L. Bagrodia and Y.-K. Tsay. An efficient algorithm for fair interprocess synchronization. Technical Report CSD-900021, Computer Science Department, UCLA, 1990.

[Cha87] A. Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(3):350–366, July 1987.

[CM86] K.M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Dij88] E.W. Dijkstra. Position paper on "fairness". *ACM SIGSOFT*, 13(2):18–20, April 1988.

[ES89] E.A. Emerson and J. Srinivasan. Branching time temporal logic. In J.W. de Bakker, W.P. de Roever, and Rozenberg G., editors, *LNCS 354: Linear Time, Branching Time and Partial Order in Logic and Models for Concurrency*, pages 123–172. Springer-Verlag, 1989.

[FHT86] N. Francez, B. Hailpern, and G. Taubenfeld. Script: A communication abstraction mechanism. *Science of Computer Programming*, 6(1):35–88, January 1986.

[FLP85] M.J. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, April 1985.

[Fra86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[Fra89] N. Francez. Cooperating proofs for distributed programs with multiparty interactions. *Information Processing Letters*, 32(5):235–242, September 1989.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, August 1978.

[Ram87a] S. Ramesh. A new and efficient implementation of multiprocess synchronization. *LNCS 259: PARLE Parallel Architecture and Languages Europe*, pages 387–401, June 1987.

[Ram87b] S. Ramesh. A new implementation of CSP with output guards. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 266–273, 1987.

[Sis84] A.P. Sistla. Distributed algorithms for ensuring fair interprocess communication. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 266–277, 1984.