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

SOME IMPOSSIBILITY RESULTS IN INTERPROCESS SYNCHRONIZATION

Yih-Kuen Tsay
Rajive L. Bagrodia

October 1989
CSD-890059

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 formally specify the problem of synchronizing asynchronous processes and two strong fairness notions in a general model. 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 supported by a grant from NSF (CCR 88 10376)

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 [Sis84, Bag89b, BS83, Bag89a, CM88]. 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 SIF is impossible for binary (and hence for multiway) interactions and SPF is impossible for multiway interactions. In another paper [BT90] we give an efficient algorithm for binary interactions with SPF. 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. From an implementation perspective, Dijkstra[Dij88] contends that fairness is a void obligation for language implementers 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 introduces the computational model and notation used in the paper. Section 3 gives a formal specification of the interaction problem along with the desired safety, liveness, and fairness properties. Section 4 gives the impossibility proofs for the unsatisfiability of some of the fairness properties.

2 Model and Definitions

2.1 Program and Computation

A *program* essentially 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. The *state* of a program is the tuple of values assumed by the program variables; an *initial* state is a state satisfying the specification of initial values of the program variables. Each *rule* is specified by 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* 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. The body of the selected rule is executed if the rule is enabled; nothing happens otherwise. The execution of a rule, enabled or disabled, 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 that each rule of the program is selected infinitely many times in a computation.

We introduce some notations:

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

α, β denote infinite sequences of rules.

σ, τ denote infinite sequences of states.

x_i denotes the i-th element of sequence x. We assume 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.

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

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

$Init(\mathcal{D})$ denotes the set of 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 $\mathcal{D}$ under the fair selection criterion. Note that $Rule^*(\mathcal{D})$ is a suffix closed set, i.e. if $\alpha \in Rule^*(\mathcal{D})$ then $\forall i : i \geq 0 :: \alpha^i \in Rule^*(\mathcal{D})$.

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

$Reach(\mathcal{D}) \equiv \{s \mid \exists s', x : s' \in Init(\mathcal{D}) \wedge x \in Pref(\mathcal{D}) :: s = (s', x)\}$ is the set of all states reachable from initial states. It follows that $Init(\mathcal{D}) \subseteq Reach(\mathcal{D})$.

$Comp(\mathcal{D}) \equiv \{\langle s, \alpha \rangle \mid s \in Init(\mathcal{D}) \wedge \alpha \in Rule^*(\mathcal{D})\}$ is the set of all possible computations of $\mathcal{D}$. Let $Comp^*(\mathcal{D})$ be the suffix closure of $Comp(\mathcal{D})$. It is easy to see that $Comp^*(\mathcal{D}) = \{\langle s, \alpha \rangle \mid s \in Reach(\mathcal{D}) \wedge \alpha \in Rule^*(\mathcal{D})\}$.

$Branch(s) \equiv \{\sigma \mid \sigma \in Comp^*(\mathcal{D}) \wedge \sigma_0 = s\}$ is the set of all possible "futures" of the state s.

2.2 Temporal Logic

Our logic is a mixture of the branching time logic in [ES89] and the first-order temporal logic in [Krö87].

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 are predicates on program states and p,q are assertions. σ is an infinite sequence in $Comp^*(\mathcal{D})$; recall that σ^i denotes its suffix σ_i,σ_{i+1}, σ_{i+2},$\cdots$. In the following definitions, the logical operators $\neg$, $\wedge$, and $\vee$ and quantifiers $\forall$ and $\exists$, when not occurring as part of an assertion, should be interpreted according to their meanings in standard predicate calculus.

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

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

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

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

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

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

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

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

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

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

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

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

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

(Notice that "$(a \ \mathit{Until}\ b) \mid \sigma$" is true if a is "currently" false (i.e. a is false at σ_0) regardless of the truth value of b; similarly for "$(a \ \mathit{Unless}\ b) \mid \sigma$." In conventional temporal logic, both assertions would be false if a and b are currently false. This modification is motivated by the ease of its usage in specifying safety properties of programs.)

A quantified assertion is interpreted as multiple occurrences of the assertion with the quantified variables being 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.* 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, assertions are often 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 mostly expressed by statements of the form "p *in* $\mathcal{D}$," where p is an assertion.

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

The following are some logical rules that will be used in this paper. They are presented without proofs and the reader is referred to [Krö87, ES89] for a discussion of their validity.

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

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

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

$$(a \ \mathit{Unless}\ (b \vee c)) \rightarrow ((a \ \mathit{Unless}\ b) \vee (a \ \mathit{Unless}\ c)) \tag{T4}$$

2.3 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 component 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 programs F and G is denoted by $F[\!]G$. Note that F and G may themselves be composite programs. 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. The *shared* variables of a program may only be shared between a process and a channel; processes do not share variables, neither do channels. Each of such shared variables is a queue where messages are appended by one sharing module and extracted by the other; the queue is called output queue for

the former module and input queue for the latter. A message is *sent* if it is extracted by a *channel* from the output queue of a *process*; a message is *received* if it is appended by a *channel* to the input queue of a *process*. We assume a channel always extracts and appends a given message in separate steps. (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.

3 Multiway and Binary Interaction Problems

The multiway interaction problem abstracts the basic issues, namely synchronization and mutual exclusion, in implementing the symmetric, nondeterministic, synchronous communication constructs of programming languages like CSP [Hoa78], Script [FHT86], and IP [Fra89] on a distributed architecture. An anthropomorphic version of this problem, called committee coordination, can be found in [CM88].

Consider a set of processes and a set of interactions defined among them. Each interaction is a nonempty subset of processes representing some synchronization activity of its members. A process can be in *active*, *idle*, or *commit* state. An active process may autonomously become idle and wait to participate in some interaction. (Note that, in general, it is impossible to determine when, or if, an active process will become idle.) An interaction is *enabled* if all of its members are idle; it is *disabled* otherwise. Only enabled interactions can be started — synchronization. An idle process will transit to commit state only after an interaction of which it is a member is started. A process can participate in at most one interaction — mutual exclusion. A process in commit state can become active only after the interaction in which it participates is terminated — another synchronization.

The binary interaction problem is a special case where each interaction has exactly two members.

3.1 Problem Specification

We adapt the UNITY format of problem specification [CM88]: Let *USER* refer to a set of asynchronous processes (including the channels) 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 $\mathcal{P}$. We use the temporal logic language introduced in section 2 to specify the properties of *USER* and $\mathcal{P}$ as well as the constraints on *OS*.

We assume processes in *USER* are numbered 1 through n and the i-th process is denoted by

$user_i$; analogously for *OS*. $p_i \equiv user_i [] os_i$ denotes the i-th 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 $user_i$, $user_j$, and $user_k$ is represented by $\{i,j,k\}$. $\mathcal{I}$ is the set of interactions defined among users; each element of $\mathcal{I}$ is a nonempty subset of $\{1,2,...,n\}$. Two interactions are said to be *conflicting* if they have at least one common member. The set of all interactions of which a process is a member is referred to as the interaction set of the process.

Each *user* and the corresponding *os* share two variables: a boolean array called **flag** and variable state which may assume the value *active*, *idle*, or *commit*. The three states of a process correspond to a user that does not want to participate in any interaction, a user that is waiting to participate in some interaction, and a user that has committed itself to a specific interaction. Each component of flag corresponds to an interaction in the user's interaction set. Interaction I is started if one of its members, say p_i, sets flag_i^I to 1 and is terminated if flag^I is set back to 0 for all members of I. We introduce some abbreviations for commonly used predicates:

$$active_i \equiv (\mathsf{state}_i = active), \text{ similary for } idle_i \text{ and } commit_i \tag{d1}$$

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

$$sync^I \equiv (\exists i : i \in I :: \mathsf{flag}_i^I = 1) \tag{d3}$$

$$E[I,J] \equiv (I \neq J \wedge I \cap J \neq \phi) \tag{d4}$$

All assertions are assumed to be universally quantified over all values of their free variables. In the remainder of this section, we formalize the behavior of a user process, the constraints imposed on the scheduler and the safety, liveness, and fairess properties desired in the composite program.

3.1.1 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, **state** is initialized to *active* and each component of **flag** to 0. An active user may autonomously transit to idle — (u1). $user_i$ becomes idle due to the execution of a sequence of rules local to $user_i$ — (u1.1) and it is always possible that, from a certain point of computation of the composite program, $user_i$ never become idle — (u1.2). An idle user may transit to commit only after some interaction in its interaction set is started — (u2) and the user transits from commit back to active only after the interaction is terminated — (u4). (u4.1) and (u4.2) state that it is always possible (though not necessary) for an interaction to be eventually terminated and all members of the interaction eventually become active. When an interaction is started, all members will eventually transit to commit — (u3). A user may not start an interaction — (u5).

$active_i$ *Unless* $idle_i$ *in USER* (u1)

$active_i \rightarrow \mathsf{E}\Diamond idle_i \ in \ user_i$ (u1.1)

$active_i \rightarrow \mathsf{E}\Box active_i \ in \ \mathcal{P}$ (u1.2)

$idle_i \ \ Unless \ \ commit_i \wedge (\exists I : i \in I :: sync^I) \ in \ USER$ (u2)

$sync^I \rightarrow \Diamond(\forall i : i \in I :: commit_i) \ in \ \mathcal{P}$ (u3)

$commit_i \ \ Unless \ \ active_i \wedge (\forall I : i \in I :: \neg sync^I) \ in \ USER$ (u4)

$sync^I \rightarrow \mathsf{E}\Diamond\neg sync^I \ in \ \mathcal{P}$ (u4.1)

$commit_i \rightarrow \mathsf{E}\Diamond active_i \ in \ \mathcal{P}$ (u4.2)

$(\mathsf{flag}_i^I = 0) \rightarrow \bigcirc(\mathsf{flag}_i^I = 0) \ in \ USER$ (u5)

3.1.2 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). As a consequence, an enabled interaction remain enabled unless itself or a conflicting interaction is started — (pp3) and an active process does not participate in any interaction — (pp4). The liveness property requires that if an interaction I is *enabled*, either I or a conflicting interaction be eventually started; in conjunction with (u3), I will eventually be disabled — (pp5).

$\neg sync^I \ \ Unless \ \ enable^I \ in \ \mathcal{P}$ (pp1)

$E[I, J] \rightarrow \neg(sync^I \wedge sync^J) \ in \ \mathcal{P}$ (pp2)

$enable^I \ Unless \ sync^I \vee (\exists J : E[I, J] :: sync^J) \ in \ \mathcal{P}$ (pp3)

$active_i \rightarrow (\forall I : i \in I :: \neg sync^I) \ in \ \mathcal{P}$ (pp4)

$enable^I \rightarrow \Diamond\neg enable^I \ in \ \mathcal{P}$ (pp5)

3.1.3 Constraints on *OS*

The only shared variables between $user_i$ and os_i are state_i and flag_i. For each os, **state** is initialized to *active* and each component of **flag** to 0 (consistent with the initialization in *USER*). An os may not change the state of a user so that the properties (u1), (u1.1), (u2), and (u4) are preserved in the composite program $\mathcal{P}$. Furthermore, an os may not terminate an interaction — (o1).

$(\mathsf{flag}_i^I = 1) \rightarrow \bigcirc(\mathsf{flag}_i^I = 1) \ in \ OS$ (o1)

3.2 Fairness

The notion of fairness in the problem specification is weak. A user is said to be *ready* if some interaction in its interaction set is enabled. The problem specification allows starvation: from a

certain point of computation, a user may become ready infinitely many times but never participate in any interaction. Also, an interaction may be enabled infinitely many times but never be started.

We are interested in two stronger fairness notions: Strong Process Fairness (SPF) and Strong Interaction Fairness (SIF). SPF asserts a user that is infinitely often ready will infinitely often participate in some interaction. SIF asserts an interaction that is enabled infinitely often will be started infinitely often. It can be shown that SPF with (u3) subsumes (pp5).

$ready_i \equiv (\exists I : i \in I :: enable^I)$ *in* $\mathcal{P}$

SPF $\equiv \Box\Diamond ready_i \rightarrow \Box\Diamond(\exists I : i \in I :: sync^I)$ *in* $\mathcal{P}$

SIF $\equiv \Box\Diamond enable^I \rightarrow \Box\Diamond sync^I$ *in* $\mathcal{P}$

4 Impossibility Results

We call an additional property *satisfiable* if there exists an *OS* satisfying the original specification such that the additional property also holds; otherwise it is *unsatisfiable*. We shall show that "SIF *in* $\mathcal{P}$" (or simply SIF) is unsatisfiable for the binary (hence for multiway) interaction problem and SPF unsatisfiable for the multiway interaction problem. We prove a property unsatisfiable by showing that, for any *OS* meeting the problem specification, *Comp*($\mathcal{P}$) always contains some computation violating the assertion in the statement of the property.

We first introduce some important properties of distributed programs.

4.1 Characteristics of Distributed Programs

Suppose $\mathcal{D}$ is a distributed program. Let Q be the composition of some of the processes in $\mathcal{D}$ and $\overline{Q}$ the composition of all processes in $\mathcal{D}$ not in Q. Note that Q and $\overline{Q}$ do not share any variables according to the definition in section 2.3. $s[Q]$ denotes the projection of 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 are 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* $(s[Q] = s'[Q]) \wedge x \in Pref(Q)$, *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 a program, the lemma follows. *End of Proof.*

Lemma 2 *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 directly. *End of Proof.*

4.2 Impossibility Proofs

In the following proofs, we consider a case where $\mathcal{P}$ has three processes p_i, p_j, and p_k, where $p_i \equiv user_i[\![os_i$ and similarly for p_j and p_k; $\mathcal{I} = \{I,J,K\}$, where $I = \{i,j\}$, $J = \{j,k\}$, and $K = \{k,i\}$. We show that SIF is unsatisfiable for this $\mathcal{P}$.

Lemma 3 *If interaction I is* enabled *and $user_k$ is* active *at some point of computation, then there exists a possible future in which interaction I is started before J and K. In other words:*

$$(enable^I \wedge active_k) \rightarrow \mathsf{E}((enable^I \ \mathit{Unless}\ sync^I) \wedge \Diamond sync^I) \ \mathit{in}\ \mathcal{P}.$$

We prove this lemma later.

Theorem 1 SIF *is unsatisfiable for the binary (and hence multiway) interaction problem.*

Proof. In any reachable state s of $\mathcal{P}$ where all users are active, we are able to construct an infinite sequence of rules $\alpha, \alpha \in Rule^*(\mathcal{P})$, such that $(\Box\Diamond enable^K \wedge \Box\neg sync^K) \mid \langle s, \alpha\rangle$, which violates SIF. ($\neg$(SIF *in* $\mathcal{P}$) $= \exists\sigma : \sigma \in Comp^*(\mathcal{P}) :: (\exists I : I \in \mathcal{I} :: \Box\Diamond enable^I \wedge \Box\neg sync^I \mid \sigma)$.) The construction proceeds in phases; 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 users become active again. To satisfy the fair selection criterion, each rule in $\mathcal{P}$ is selected at least once in each phase. Starting from a state s_0 where all processes are active, each phase goes as follows:

Stage 1: Apply (u1.1) first to $user_i$ then to $user_j$ (or the other way around) to obtain a sequence x_1 consisting of rules of $user_i$ and $user_j$ such that $(idle_i \wedge idle_j)$ at (s_0,x_1). From (d2), $enable^I$ at (s_0,x_1). Those rules of $user_i$ and $user_j$ not selected can be arranged in arbitrary order to form a sequence x_2. Let $s_1 = (s_0,(x_1;x_2))$. As $(x_1;x_2)$ contains rules from only $user_i$ and $user_j$, due to (u2) and (u5), no interaction is started hence $enable^I$ at s_1; and, since $active_k$ at s_0 and $(x_1;x_2)$ does not contain any rules in $user_k$, from lemma 2, we get $active_k$ at s_1.

Stage 2: Given $(enable^I \wedge active_k)$ at s_1, from lemma 3, there exists a sequence y_1 of rules in $\mathcal{P}$ such that $sync^I$ at (s_1,y_1) and $\neg sync^J \wedge \neg sync^K$ 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 os_i or os_j, and $\neg sync^I$ at (s_1,y_0). From (pp1), $enable^I$ at (s_1,y_0). Assume that $active_k$ at all states in $\langle s_1, y_1\rangle$. According to (u1.1), there exists a sequence y_2

of rules in $user_k$ such that $idle_k$ at $(s_1,(y_0;y_2))$, hence $(enable^I \wedge enable^J \wedge enable^K)$ at $(s_1,(y_0;y_2))$. (If $idle_k$ at some state in $\langle s_1, y_1\rangle$, then y_2 is simply the empty sequence.)

From lemma 2 (replacing s and s' in the lemma by (s_1,y_0), Q by $os_i[\!]os_j$, $\overline{Q}$ by os_k, and x by y_2), $(s_1,y_0)[os_i[\!]os_j] = ((s_1,y_0),y_2))[os_i[\!]os_j]$, which is to say $(s_1,y_0)[os_i[\!]os_j] = (s_1,(y_0;y_2))[os_i[\!]os_j]$.

From lemma 1 (replacing s by (s_1,y_0), s' by $(s_1,(y_0;y_2))$, Q by $os_i[\!]os_j$, and x by r),

$((s_1,y_0),r)[os_i[\!]os_j] = ((s_1,(y_0;y_2)),r)[os_i[\!]os_j]$. As $((s_1,y_0),r) = (s_1,(y_0;r))$ and $y_1 = (y_0;r)$, we get $(s_1,y_1)[os_i[\!]os_j] = (s_1,(y_0;y_2;r))[os_i[\!]os_j]$. Let $s_2 = (s_1,(y_0;y_2;r))$.

Since $sync^I$ at (s_1,y_1), the preceding statement implies $sync^I$ at s_2. (1)

Also, from lemma 2, $idle_k$ at $(s_1,(y_0;y_2))$ implies $idle_k$ at s_2. (2)

Those rules of $\mathcal{P}$ except $user_i$ and $user_j$, not selected in the sequence $(y_0;y_2;r)$ can form an arbitrary sequence y_3. As y_3 does not contain any rules of $user_i$ or $user_j$, from lemma 2 and (1), $sync^I$ at all states in $\langle s_2, y_3\rangle$, which implies $\neg sync^K$ at all states in $\langle s_2, y_3\rangle$, due to (pp2). According to (u2), (o1), and (2), $idle_k$ at (s_2,y_3). Let $s_3 = (s_2,y_3)$. So,

$(sync^I \wedge idle_k)$ at s_3. (3)

Stage 3: Interaction I was started in the previous stage at state (s_1,y_1). By virtue of (u3), $user_i$ and $user_j$ transit from idle to commit; by (u4.1), interaction I is terminated, and by (u4.2), $user_i$ and $user_j$ transit from commit to active. Similar to Stage 1, apply (u1.1) to $user_j$ such that $user_j$ becomes idle again; however $user_i$ remains active. Let z be the corresponding sequence and $s_4 = (s_3,z)$. $idle_j$ at s_4 and $active_i$ at s_4. From (3) and lemma 2, $idle_k$ at s_4. So, $(enable^J \wedge active_i)$ at s_4.

Stage 4: Similar to Stage 2, interaction J is started and, similar to Stage 3, both p_j and p_k eventually become active. Let w be the sequence and $s_5 = (s_4,w)$. All processes are active at s_5.

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 $(\Box\Diamond enable^K \wedge \Box\neg sync^K) \mid \langle s_0,\alpha\rangle$. Each rule in $\mathcal{P}$ is selected at least once either in Stage 1 or Stage 2, so $\alpha \in Rule^*(\mathcal{P})$.

End of Proof.

Theorem 2 SPF *is unsatisfiable for the multiway interaction problem.*

Proof. Add one process p_l to $\mathcal{P}$ and change K to $\{k,i,l\}$. We obtain an instance of multiway interaction problem.

At some point of computation, assume that p_l transits to 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.

The following is the proof of lemma 3:

Assume the contrary; from definitions (A6.3) and (A5.2),

$\exists \sigma : \sigma \in Comp^*(\mathcal{P}) :: (enable^I \wedge active_k) \wedge \mathsf{A}\neg((enable^I \ Unless \ sync^I) \wedge \Diamond sync^I) \mid \sigma.$

Fix σ and expand the assertion in the above statement using (T2) and (A5.1) to get

$\forall \tau : \tau \in Branch(\sigma_0) :: enable^I \wedge active_k \wedge \neg((enable^I \ Unless \ sync^I) \wedge \Diamond sync^I) \mid \tau.$ (1)

For any such τ, elaborating the third conjunct using (A4.2), we get

$\neg(enable^I \ Unless \ sync^I) \vee \Box\neg sync^I \mid \tau.$ (2)

Since there are only three interactions I, J and K, from (pp3),

$enable^I \ Unless \ (sync^I \vee sync^J \vee sync^K) \mid \tau.$ (3)

$(enable^I \ Unless \ sync^I) \vee (enable^I \ Unless \ (sync^J \vee sync^K)) \mid \tau$, from (T4).

From this, (2) implies that

$(enable^I \ Unless \ (sync^J \vee sync^K)) \vee \Box\neg sync^I \mid \tau.$ (4)

In other words, interaction I will never be started or will be disabled due to J or K being started.

$\exists \tau' : \tau' \in Branch(\sigma_0) :: \Box active_k \mid \tau'$, from (u1.2) and (A5.2). (5)

$\Box(active_k \rightarrow (\neg sync^J \wedge \neg sync^K)) \mid \tau'$, from (pp4) and (P1).

$\Box(\neg sync^J \wedge \neg sync^K) \mid \tau'$ $(= (\Box\neg sync^J \wedge \Box\neg sync^K) \mid \tau')$, from (5), (T3), and (T1). (6)

$\Box enable^I \vee \Box\neg sync^I \mid \tau'$, from (4), (6), and (A7.2).

$\neg\Box enable^I \mid \tau'$, from (pp5).

$\Box\neg sync^I \mid \tau'$, from the above two statements.

$enable^I \wedge \Box\neg sync^I \wedge \Box\neg sync^J \wedge \Box\neg sync^K \mid \tau'$, from the preceding statement, (1), and (6).

$enable^I \wedge \Box\neg(sync^I \vee sync^J \vee sync^K) \mid \tau'$, from (T1).

With this and (3), we conclude that $\Box enable^I \mid \tau'$, which violates (pp5). *End of Proof.*

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, Computer Science Department, UCLA, June 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. *JACM*, 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.

[Krö87] F. Kröger. *Temporal Logic of Programs.* Springer-Verlag, 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.