Fairness and Hyperfairness in Multi-party Interactions

Paul C. Attie[1,2] *Nissim Francez*[2,3] *Orna Grumberg*[3]

1. Department of Computer Sciences, The University of Texas at Austin, USA
2. Microelectronics and Computer Technology Corporation, Austin, Texas, USA
3. Computer Science Department, Technion, Haifa, Israel

August 7, 1992

Abstract

In this paper, a new fairness notion is proposed for languages with *multi-party interactions* as the sole interprocess synchronization and communication primitive. The main advantage of this fairness notion is the elimination of starvation occurring solely due to race conditions (i.e., ordering of independent actions). Also, this is the first fairness notion for such languages which is fully-adequate with respect to the criteria presented in [2]. The paper defines the notion, proves its properties, and presents examples of its usefulness.

1 Introduction

Fairness is one of the most important classes of *liveness* properties employed by languages for nondeterministic, concurrent and distributed programs and by their underlying models of computation. This importance stems from the semantic intricacies of constructs in such languages, which makes the verification of typical progress properties (e.g., eventual response to a request for service) difficult. One source of difficulty is the phenomenon of *conspiracies*. A conspiracy occurs if a request for service is never granted because conflicting requests intermittently engage some of the needed resources. In this paper, we investigate the prevention of conspiracies through fair scheduling. We propose a new fairness notion that prevents a conspiracy whenever it is due solely to "race conditions", i.e., whenever its prevention involves delaying some requests and proceeding with independent, non-conflicting requests.

An appropriate setting for the study of conspiracies and their prevention is that of concurrent programs expressed in programming languages with *multi-party interactions* as their primitive construct for interprocess synchronization and communication. In our opinion, a multi-party interaction is more abstract and higher level than the more commonly used point-to-point communication which is usually expressed by message-passing primitives such as *send* and *receive* (either asynchronous, or *CSP*-like handshaking) or by remote procedure calls (e.g., *ADA*-like *rendezvous*). A single multi-party interaction hides the details of its implementation by several point-to-point communication operations. Among other things, the ordering of point-to-point communication operations is abstracted away, and so high level designs can be produced which do not contain such low level details. Several such multi-party interaction constructs have been proposed, e.g., *scripts* [14], *joint actions* [5], *shared actions* [25], *compacts* [9] and *teams* and *interactions* [15]. In addition, the *handshakes* in Hoare's algebraic version of *CSP* [20] are also multi-party synchronizations, though this fact is not stressed there. It is our view that such constructs will become more frequently used as more and more complicated concurrent programs are designed, especially since many implementations of multi-party interactions have appeared [4, 6, 8, 10, 11].

We deal with both serialized and overlapping semantics. In serialized semantics, interaction execution is atomic, i.e., one interaction is executed at a time, and parallelism is modeled by the nondeterministic interleaving of interaction executions. In overlapping semantics, interaction execution has finite (nonzero) duration, so executions of non-conflicting interactions can overlap in time. Parallelism is modeled by the nondeterministic interleaving of interaction fragments, which, roughly speaking, are events that represent the start and finish of interaction executions. As is well known by now for binary interactions (i.e., interactions with exactly two participants) [2, 6, 7, 16, 18], natural fairness properties, which are satisfied in a *serialized* execution (one interaction at a time), need not hold in an *overlapping* execution. This also holds, of course, for general multi-party interactions.

Within the setting of multi-party interactions, a conspiracy occurs if an interaction is never enabled for execution because conflicting interactions intermittently engage some of the needed participants, i.e., participants are viewed as resources which are required for the enablement of an interaction (this situation occurs in both serialized and overlapping semantics). Our fairness notion prevents "race conditions" by delaying some interactions and proceeding with independent, non-conflicting interactions (this is made more precise in the sequel). Our notion also satisfies the three semantic criteria posed in [2] for fairness notions, (namely *feasibility*, *liveness enhancement*, and *equivalence*

robustness, which are briefly explained in this paper) and is the first fairness notion for multi-party interactions that does so. The existence of such a fairness notion was presented as an open problem by Pnueli [24]. This problem is positively solved in this paper, though it is not clear at this time whether the solution is the most general one.

The rest of the paper is organized as follows. In Section 2, a mini-language, *IP*, (an abstraction of Raddle [15]) is presented. Both its serialized and overlapping semantics are given. In Section 3 the concept of *conspiracy resistance* is presented, in terms of which hyperfairness is defined, and examples are presented which rely on hyperfairness for their required liveness properties. In Section 4 an *explicit scheduler* transformation [3, 22] for hyperfairness is proposed and proven to be *faithful,* i.e., it generates all and only hyperfair computations of a given program. In Section 5 our conclusions are presented.

2 The Language IP (Interacting Processes)

In this section we present a simple mini-language, called *IP* (*Interacting Processes*), first introduced in [17]. The language is an abstraction and simplification of programming languages containing the multi-party interaction, and is suitable for focusing on fair conflict-resolutions among interactions, omitting other features found in similar, but more elaborate languages. Its main feature is the usage of multi-party interactions as *guards,* thereby generalizing both Dijkstra's original guarded commands language [12], which has only boolean guards, and *CSP* [19], which uses synchronous binary communication operations as guards. Further results regarding the proof-theory (of partial-correctness) of multi-party interactions, expressed via the *IP* language, are reported in [17].

A program $P :: [\, P_1 \| \ldots \| P_n \,]$ consists of a *concurrent composition* of $n \geq 1$ (fixed n) *processes*, having *disjoint* local states (i.e., no shared variables). A *process* P_i, $1 \leq i \leq n$, consists of a statement S, where S may take one of the following forms:

dummy statement - *skip*: A statement with no effect on the state.

assignment - $x := e$: The variable x is local to P_i and e is an expression over P_i's local state.

interaction - $a\,[\bar{v} := \bar{e}]$: Here a is the *interaction name* and $\bar{v} := \bar{e}$ is an optional parallel assignment constituting a *local interaction body.* Process P_i is a *participant* of interaction a. All variables in $\bar{v}$ are local to P_i and pairwise distinct. The expressions $\bar{e}$ may involve variables *not local* to P_i (belonging to other participants of interaction a). The participants of interaction a, denoted PA_a, is the set of all processes that syntactically refer to a. Interaction a is *readied* by process P_i, if control of P_i has reached a point where executing a is one of the possible continuations. Interaction a is *enabled* if and only if *all* of its participants have readied it. It can be executed only if it is enabled. Thus, an interaction synchronizes all of its participants. Execution of an interaction consists of the parallel execution of the local interaction bodies contained in all participant processes. Variables within $\bar{e}$ return the value that they held immediately before execution of interaction a. Upon termination of a local interaction body a participating process resumes its local thread of control (i.e., no synchronization at the end of an interaction). Note that if the body $\bar{v} := \bar{e}$ is empty (so the interaction appears as $a\,[\,]$) for some participating process, the effect of the interaction on that process is pure synchronization.

sequential composition - $S_1;\ S_2$: First S_1 is executed. If and when it terminates, S_2 is executed. We freely use $S_1;\ \ldots\ ;\ S_k$ for any $k \geq 2$.

nondeterministic selection - $[\mathop{[]}_{k=1,m} B_k; a_k\,[\bar{v}_k := \bar{e}_k] \rightarrow S_k]$: Here $B_k; a_k\,[\bar{v}_k := \bar{e}_k]$ is a *guard*, composed of two parts. The part B_k is a boolean expression over the local state of P_i. The part $a_k\,[\bar{v}_k := \bar{e}_k]$ is an *interaction guard*. S_k is any statement. When a nondeterministic selection statement is evaluated in some state, the k'th guard is *open* if B_k is true in that state (note that the interaction a_k is *readied* by P_i at that state). The guard is *enabled* if and only if it is open and the interaction a_k is enabled. Executing the statement involves the following steps: evaluation of all boolean parts to determine the collection of open guards. If this collection is empty the statement *fails*. Otherwise an enabled guard is passed (simultaneously with the execution of all the other matching local interaction bodies of a_k in the other participating parties) and then S_k is executed. In case there are open guards, but none is enabled, execution is *blocked* (possibly forever) until some open guard is enabled.

nondeterministic iteration - $*[\mathop{[]}_{k=1,m} B_k; a_k\,[\bar{v}_k := \bar{e}_k] \rightarrow S_k]$: Similar to the nondeterministic selection, but execution terminates once no open guards exist, and execution of the whole statement is repeated after each execution of a guarded command.

Note that nested concurrency is excluded by the above definition, since it is orthogonal to the issue under investigation. We now turn to formal definitions of the operational semantics, based on Plotkin's transition scheme [23]. We define two different semantics: *serialized* and *overlapping* (compare with a similar distinction in [7] and [18]). The former is better suited for correctness proofs, while the latter captures better the behavior induced by typical implementations. The difference between the serialized and overlapping semantics is, that in the former each program action is represented by a single transition, while in the latter every program action is represented by a number of transitions (referred to as *action fragments*), one causing the effect of the program action on the state, the rest releasing the participants. The proposed hyperfairness handles conspiracies under both semantic definitions. Throughout we assume some *interpretation* over which computations occur, and leave it implicit.

2.1 Serialized Semantics

The central characteristic of this semantics is that local actions and interactions take place one at a time. A *configuration* $< [S_1 \| \ldots \| S_n],\ \sigma >$ consists of a concurrent program and a global state. The global state is a mapping from program variables to values in the domain of the interpretation. A configuration represents an intermediate stage in a computation where S_i is the rest of the program that process P_i has still to execute (sometimes referred to as its syntactic continuation), while σ is the *current* state at that stage. We stipulate (for facilitating the definition) an *empty* continuation E (not a program in the language) satisfying the identities $S;\ E = E;\ S = S$ for every S. A configuration $< [E \| \ldots \| E],\ \sigma >$ is a *terminal* configuration. For a state σ, we use the usual notions of a *variant* $\sigma[c/x]$ and $\sigma[\bar{c}/\bar{x}]$, obtained from σ by changing the value of x to c ($\bar{x}$ to $\bar{c}$, respectively) and preserving the values of all other variables. We use $\sigma(e)$ to denote the value of an expression e in a state σ. We use PA_a to denote the set of all processes that participate in interaction a, i.e., all processes which contain a construct "$a\,[\bar{v} := \bar{e}]$" within their body. We now define the (*serialized*) *transition* relation "$\rightarrow$" among configurations. Note that the transitions

given by "$\rightarrow$" are atomic.

$$< [S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_n],\ \sigma > \ \rightarrow < [S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_n],\ \sigma > \quad (1)$$

for any $1 \leq i \leq n$, iff $S_i = skip$, or $S_i = *[\underset{j=1,n_i}{[]} B_j; a_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$ and $\neg \underset{j=1,n_i}{\vee} B_j$ holds in σ.

$$< [S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_n],\ \sigma > \ \rightarrow < [S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_n],\ \sigma[\sigma(e)/x] > \quad (2)$$

for any $1 \leq i \leq n$, iff $S_i \ = \ (x := e)$.

$$< [S_1 \| .. \| S_{i_1-1} \| S_{i_1} \| S_{i_1+1} \| .. \| S_{i_k-1} \| S_{i_k} \| S_{i_k+1} \| .. \| S_n], \sigma > \rightarrow < [S_1 \| .. \| S_{i_1-1} \| S'_{i_1} \| S_{i_1+1} \| .. \| S_{i_k-1} \| S'_{i_k} \| S_{i_k+1} \| .. \| S_n], \sigma' > \quad (3)$$

iff the following holds: There is an interaction a with a set of participants $PA_a = \{i_1, \ \ldots \ , i_k\}$ (for some $1 \leq k \leq n$), and for every $i \in PA_a$ one of the following conditions holds:

(a) $S_i \ = \ a[\bar{v}_i := \bar{e}_i]$ and $S'_i = E$

(b) $S_i \ = \ [\underset{j=1,n_i}{[]} B_j; a_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$ and there exists some j, $1 \leq j \leq n_i$, s.t. B_j holds in σ, $a_j = a$ and $S'_i = T_j$

(c) $S_i \ = \ *[\underset{j=1,n_i}{[]} B_j; a_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$ and there exists some j, $1 \leq j \leq n_i$, s.t. B_j holds in σ, $a_j = a$, $S'_i = T_j;\ S_i$.

Finally, for all these cases, $\sigma' = \sigma[\sigma(\bar{e})/\bar{v}]$, with $\bar{v} = \underset{i \in PA_a}{\cup} \bar{v}_i$, $\bar{e} = \underset{i \in PA_a}{\cup} \bar{e}_i$, where $\cup$ denotes a union operation which is ordered by process index, e.g., if $\bar{v}_1, \bar{v}_2, \bar{e}_1, \bar{e}_2$ are $size, q, size + 1, append(q, z)$ respectively, then $\sigma' = \sigma[\sigma(size + 1)/size, \sigma(append(q, z))/q]$.

If

$< [S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_n],\ \sigma > \rightarrow < [S'_1 \| \ldots \| S'_i \| \ldots \| S'_n],\ \sigma' >$

then

$$< [S_1; T_1 \| \ldots \| S_i;\ T_i \| \ldots \| S_n; T_n],\ \sigma > \rightarrow < [S'_1; T_1 \| \ldots \| S'_i;\ T_i \| \ldots \| S'_n; T_n],\ \sigma' > \quad (4)$$

where some of the S'_i may be identical to the corresponding S_i. Note that this serialized semantics does impose synchronization at the end of an interaction. For this semantics, we now define the following notions.

Definitions:

(1) A (*serialized*) *computation* π of P on σ is a maximal (finite or infinite) sequence of configurations C_i, $i \geq 0$, such that:

(a) $C_0 \ = < P, \sigma >$.

(b) For all $i \geq 0$, if C_i is not the last configuration in π, then $C_i \rightarrow C_{i+1}$.

(2) The computation π *terminates* iff it is finite and its last configuration is terminal; π *deadlocks* iff it is finite and its last configuration is *not* terminal.

(3) An interaction a is *enabled* in a configuration C iff C has one of the forms in clause (3) of the definition of '$\rightarrow$'

and all the conditions are satisfied for a.

(4) Two interactions a_1 and a_2 are in *conflict* in a configuration C iff both interactions are enabled in C and they have non-disjoint sets of participants, i.e., $PA_{a_1} \bigcap PA_{a_2} \neq \emptyset$.

(5) An *action* is an interaction, a local assignment, or *skip*.

(6) Two actions a_1 and a_2 are *independent* iff they have no common participant.

(7) Two computations are equivalent iff they differ only in the order of execution of independent actions.

(8) The *interaction trace* of a computation is the sequence of interaction names of the interactions that were executed in the computation. Often, the interaction trace suffices to uniquely identify a computation.

2.2 Overlapping Semantics

The main characteristic of this semantics is that it is *not* an interleaving of actions. Rather, actions have (unspecified) *duration,* so that an action can start while another action is in progress (obviously, a non-conflicting one). This is still represented as an interleaving of so called atomic *action fragments* which are of a finer grain of atomicity than that of program actions.

In order to allow the treatment of this semantics within the same transitional framework, we augment a configuration with an additional boolean array (of size n, the number of processes), called the *readiness* state and denoted by ρ. If $\rho[i] = true$, $1 \leq i \leq n$, in a configuration C, this means that process P_i is currently ready to engage in an action (either local or with other participants); otherwise, P_i is *engaged* in some action. In our semantics, the "real" local-state transformation is instantaneous and resets the readiness bit of the acting process(es). However, a participating process can not engage in any other action (including a local one), until another transition, setting the readiness bit, has taken place. Also, in this semantics participants of an interaction are synchronized upon entrance, but are not synchronized upon exit of the interaction. We denote the resulting transition relation by '$- - - \rightarrow$'. Its defining clauses are similar to the ones in the serialized case, with the following changes.

(a) A state transformation transition can only take place if the readiness bits of all participants are *true.*

(b) In the resulting configuration, the readiness bits of all participants are *false.*

(c) The following atomic transition is added
$< [S_1 \| .. \| S_n], \sigma, (\rho[1], .., \rho[i-1], false, \rho[i+1], .., \rho[n]) > - - - \rightarrow < [S_1 \| .. \| S_n], \sigma, (\rho[1], .., \rho[i-1], true, \rho[i+1], .., \rho[n]) >$
for every $1 \leq i \leq n$.

An *overlapping* computation π of P is defined similarly to the serialized case, except that the transition $- - - \rightarrow$ replaces $\rightarrow$, and the following conditions are added:

(a) In the initial configuration $\rho[i] = true$ for all $1 \leq i \leq n$.

(b) In a terminal configuration $\rho[i] = true$ for all $1 \leq i \leq n$.

(c) In an infinite computation, $\rho[i] = true$ infinitely-often, for every $1 \leq i \leq n$ (i.e., every action terminates within a finite time).

Likewise, the concepts of deadlock, termination and conflict are defined similarly to the serialized case.

We say that process P_i *readies* an interaction a in a configuration C iff P_i satisfies one of the conditions imposed in (a) - (c) in clause (3) of the definition of '$\rightarrow$' and $\rho[i] = true$. We denote this property by $ready_i(a)$. The interaction a is *enabled* in a configuration C iff it is jointly readied in C by all its participants, i.e., $\bigwedge_{i \in PA_a} ready_i(a)$ holds in C.

Note again that the difference between the serialized and overlapping semantics is, that in the former each program action is represented by a single transition, while in the latter every program action is represented by a number of transitions (referred to as *action fragments*), one causing the effect of the program action on the state, the rest releasing the participants. The notion of independence is naturally extended also to action fragments and their representing transitions.

To simplify the notation, we refer only to programs in a normal form similar to the one defined in [1] for CSP programs. In this form, each process consists of one main nondeterministic iteration (after some initialization assignments). The interaction-bodies serving as guards in these iterations are referred to as *top-level* interaction-bodies. A *top-level interaction* is an interaction all of whose bodies are top-level.

3 Fairness and Hyperfairness for Multi-Party Interactions

We briefly review known fairness notions for multi-party interactions and then suggest a new fairness notion, called *hyperfairness,* having some desirable properties which the previous notions lack. The main additional advantage is that the property of *conspiracy resistance* (to be explained below) is exploited.

Following the distinctions made in [7, 13, 18, 21] (see also [16], chapter 5), the classification of fairness notions for multi-party interactions presented in [2] is along two orthogonal directions.

(a) *The subject of fairness*: a process, a (fixed) group of processes and a *specific* interaction among a group of processes.

(b) *The level of fairness*: The three common levels are unconditional, weak and strong, which differ in the enablement conditions required to guarantee eventual execution of the subject action. See [16] for a detailed description of these levels and other variants of fairness. We summarize this classification for the IP language in Table 1.

Remarks on Table 1:

(1) This notion is only meaningful in a context where processes (resp. groups, interactions) are always enabled.

(2) A process is enabled if (any) one of its actions is enabled.

level/subject	Unconditional	Weak	Strong
Process	In an infinite computation every process interacts infinitely often (in any interaction). See remark 1.	In an infinite computation every continuously enabled process interacts infinitely often. See remark 2.	In an infinite computation every infinitely-often enabled process interacts infinitely often.
Group	In an infinite computation every interacting group interacts infinitely often (in any of its mutual interactions). See remarks 1 and 3.	In an infinite computation every continuously enabled interacting group interacts infinitely often.	In an infinite computation every infinitely-often enabled interacting group interacts infinitely often.
Interaction	In an infinite computation every (specific) interaction occurs infinitely often. See remark 1.	In an infinite computation every continuously enabled interaction occurs infinitely often.	In an infinite computation every infinitely-often enabled interaction occurs infinitely often.

Table 1: Fairness notions for multi-party interactions

(3) Note that a given interacting group may have several different interactions which have that group as the set of participants.

As noted in [2], none of the fairness notions in Table 1 are fully adequate (in a sense defined there and reviewed in the next section) for multi-party interactions. We now propose a fairness notion that *is* adequate.

The main idea can be informally described as follows. A *conspiracy* (w.r.t. an interaction a) occurs if PA_a can be partitioned into two (disjoint) subsets $PA_a^{\mathcal{I}} \cup PA_a^{\mathcal{P}}$, such that all members of $PA_a^{\mathcal{I}}$ are continuously ready to participate in the interaction a, while members of $PA_a^{\mathcal{P}}$ ready a infinitely-often, but *not at the same time*, i.e., not in the same configuration. Thus, in such an infinite computation a is *not* infinitely-often enabled, and need not eventually occur by any of the previously mentioned fairness notions. If the conspiracy is due to the inherent semantics of the program (e.g., see Figure 4), then there is no way to guarantee the enablement of interaction a. If, however, the conspiracy is due to the interleaving of independent actions, or action fragments, (e.g., see Figure 1) then it should be possible to control the scheduling in such a way as to eventually cause the enablement of a. This is exactly the behavior which hyperfairness enforces, as described in the succeeding paragraphs.

We wish to ensure that if the members of $PA_a^{\mathcal{P}}$ can infinitely often ready a *independently* of $PA_a^{\mathcal{I}}$ and of each other, then a will eventually be executed. This is achieved as follows. It is convenient to use one of the well known fairness notions (given in Table 1 above) to ensure actual execution. What remains then, is to ensure that the enablement condition of this "underlying" fairness notion is met. Since members of $PA_a^{\mathcal{P}}$ ready a independently of $PA_a^{\mathcal{I}}$ and of each other, they can be successively "frozen" in a state in which they ready a (thereby joining $PA_a^{\mathcal{I}}$), and when they are all "frozen" there, a is enabled. If a is enabled sufficiently often, the underlying fairness notion will ensure the eventual execution of a. In this paper, we chose the underlying fairness notion to be strong interaction fairness, and in the sequel, we use "fairness" to denote strong interaction fairness, unless otherwise stated. We now turn to a formal presentation of these ideas.

Definition (Pnueli).

(1) A process P_i is *insistent* on an interaction a from a certain point in a computation π iff it continuously readies a

in π from that point onwards [24].

(2) A process P_i is *persistent* on an interaction a in a computation π iff it readies a infinitely-often in π.

Definition.
Let P be a program in normal form, a an interaction in P, and A an arbitrary subset of PA_a. The (a, A)-*derived* program $P_{a,A}$ is obtained from P by replacing with $false$ the local guard of every top level interaction body b (where $b \neq a$) in every process $P_i \in A$.

In other words, when any processes in A has a continuation from the top level, it is prevented from participating in any interaction other than a.

Definition (Conspiracy Resistance).
An interaction a is *conspiracy resistant* in a program P iff for every fair computation π the following condition holds:

> Let π_1 be any finite prefix of π with final configuration $C = < S, \sigma >$, and let $PA_a^{\mathcal{I}}$ be the set of all the participants of a that ready a in C. Then, for every fair computation π_2 of the $(a, PA_a^{\mathcal{I}})$-derived program $S_{a,PA_a^{\mathcal{I}}}$ obtained from S, (starting in state σ), there exists a participant $P_j \in (PA_a - PA_a^{\mathcal{I}})$ such that P_j eventually readies a along π_2.

We refer to this as each process in $PA_a^{\mathcal{I}}$ being a–$frozen$, and to P_j as *independently readying* the interaction a. Thus, conspiracy-resistance may be achieved whenever the participants of a can ready a *independently* of each other. The implied independence manifests itself by the fact that a-freezing an arbitrary subset of the participants of a does not prevent the eventual readiness for a of yet another participant. A scheduler that gradually "freezes" more and more participants of interaction a once such participants ready a ensures the eventual enablement of a. The key point here is that conspiracy resistance guarantees that this freezing strategy will never cause a deadlock, regardless of the order in which processes are frozen. Note that in the above definition $PA_a^{\mathcal{I}} = \emptyset$ is possible. We now come to the definition of our central notion.

Definition (Hyperfairness).
If P is an IP program in which every top-level interaction is conspiracy-resistant, then an infinite computation π of P is *hyperfair* iff π is fair and every top-level interaction is infinitely-often enabled in π. Also, every finite computation of P is hyperfair.
If P is an IP program in which not every top-level interaction is conspiracy-resistant, then every computation π of P is hyperfair.

It is important to notice an essential difference between the fairness notions considered so far on the one hand, and hyperfairness on the other hand. The former imply eventual execution if an appropriate enablement condition is satisfied. Hyperfairness implies eventual enablement (and subsequent execution due to the underlying fairness) if the condition of conspiracy resistance is satisfied. Note that enablement and conspiracy resistance are different types of conditions. An enablement condition for a particular interaction a of a particular program P may hold in some computations (π) of P but not in others, i.e., it is a function of a, P, and π, whereas conspiracy resistance is not a

$PHIL :: [p_1 \| \ldots \| p_n \| f_1 \| \ldots \| f_n]$ where

$p_i :: s_i := \ 't';$
$\quad *[\ s_i = \ 't' \rightarrow s_i := \ 'h'$
$\quad\quad [\!]$
$\quad\quad s_i = \ 'h';\ get\text{–}forks_i\,[s_i := \ 'e'] \rightarrow give\text{–}forks_i\,[s_i := \ 't']$
$\quad\quad],$

$f_i :: *[\ get\text{–}forks_i\,[\,] \rightarrow give\text{–}forks_i\,[\,]$
$\quad\quad [\!]$
$\quad\quad get\text{–}forks_{i+1}\,[\,] \rightarrow give\text{–}forks_{i+1}[\,]$
$\quad\quad].$

Figure 1: A starvation free solution of the dining philosophers problem in hyperfair *IP*

property of individual computations, i.e., it is a function only of a and P.

With any of the previous fairness notions, if a (top level) interaction a does not satisfy the enablement condition, e.g., if a is never enabled along a computation π, then π is vacuously fair with respect to a. However, this is not the case with hyperfairness. If all top level interactions are conspiracy resistant, then π is not hyperfair with respect to interaction a *because* a is not enabled at any point of π. Note that if a different underlying fairness notion were used, e.g., weak interaction fairness, then a different hyperfairness notion would be required, namely one that ensures the required enablement conditions for weak interaction fairness. We now present several examples which illustrate hyperfairness. These examples will be analyzed using serialized semantics.

Example 1 (dining philosophers). A typical problem which can be solved by the introduction of hyperfairness is the famous *dining philosophers* problem. A solution formulated in the *IP* language is presented in Figure 1. In this example, when the i'th philosopher p_i is *hungry* ($s_i = \ 'h'$) it can *eat* ($s_i = \ 'e'$) by interacting in the three-party interaction $get\text{–}forks_i$ (together with the i'th fork f_i and the $(i-1)$'th fork f_{i-1}). In this and all other dining philosopher examples, all operations on indices are cyclic. After p_i finishes eating, it becomes *thinking* ($s_i = \ 't'$) by interacting with the same forks once again in the $give\text{–}forks_i$ interaction.

This program displays conspiratorial behavior. Let $n = 4$ (i.e., 4 philosophers) and suppose that p_2 is in a *hungry* state and waits (insistently!) for the interaction $get\text{–}forks_2$ to be executed. In order for that interaction to be enabled, it has to be readied simultaneously by both fork-processes f_1 and f_2. Consider the following interaction trace π of a serialized computation

$$\pi \ = \ get\text{–}forks_1\ get\text{–}forks_3\ (\,give\text{–}forks_1\ get\text{–}forks_1\ give\text{–}forks_3\ get\text{–}forks_3\,)^\omega$$

It is clear that $get\text{–}forks_2$ is never enabled in π because $get\text{–}forks_1$ and $get\text{–}forks_3$ alternately engage f_1 and f_2 respectively. Note that the conspiracy against $get\text{–}forks_2$ in π is due to "unfortunate" scheduling which prevents f_1 and f_2 from readying $get\text{–}forks_2$ simultaneously. This is an example of conspiracy occurring in the serialized semantics, since π is a serialized computation. Now p_i is insistent on $get\text{–}forks_i$ (once $s_i = \ 'h'$), and the two fork processes f_i and f_{i-1} ready $get\text{–}forks_i$ independently of each other and of p_i, so the interaction $get\text{–}forks_i$ (for every

$PHIL :: [p_1 \| \ldots \| p_n \| f_1 \| \ldots \| f_n \| c]$ where

$p_i :: \ldots$ as in Figure 1 $\ldots$

$f_i :: clean_i := true;$
*[$clean_i$; $get\text{–}forks_i$ [ ] $\rightarrow$ $give\text{–}forks_i$ [$clean_i := clean\text{–}test_i$]
[]
$clean_i$; $get\text{–}forks_{i+1}$ [ ] $\rightarrow$ $give\text{–}forks_{i+1}$[$clean_i := clean\text{–}test_i$]
[]
$\neg clean_i$; $cleaning_i$ [$clean_i := true$] $\rightarrow$ $skip$
],

$c :: \{cleaning\ service\} * [\underset{i=1,n}{[]} cleaning_i [\,] \rightarrow skip]$

Figure 2: Dining philosophers with a cleaning service

$1 \leq i \leq n$) is conspiracy-resistant. As all top level interactions are conspiracy resistant, by hyperfairness, $get\text{–}forks_i$ must be enabled infinitely often, and so π will be excluded as an illegal (i.e., not hyperfair) computation. Note that in the interaction trace of the equivalent computation

$$\pi' \;=\; get\text{–}forks_1\ get\text{–}forks_3\ (\ give\text{–}forks_1\ give\text{–}forks_3\ get\text{–}forks_1\ get\text{–}forks_3\ )^\omega$$

in which the $get\text{–}forks_1$ interaction is commuted with the independent $give\text{–}forks_3$ interaction, no conspiracies occur, and $get\text{–}forks_2$ is infinitely often enabled (it is enabled immediately after $get\text{–}forks_3$ has been executed). But π' is also not hyperfair, because $get\text{–}forks_2$ is infinitely often enabled but never executed, and so π' is not fair.

Example 2 (dining philosophers with fork cleaning). In the previous example, the two forks readied the $get\text{–}forks_i$ interaction unconditionally. Here is a somewhat strengthened example (Figure 2), where the forks have boolean conditions and additional interactions with a *cleaning* process c, which interacts with a non-clean fork to clean it. Here $clean\text{–}test_i$ represents some hidden condition, returning a boolean value representing the state of cleanliness of a fork. We assume that $clean\text{–}test_i$ is guaranteed to return $'false'$ infinitely often if invoked infinitely often, since otherwise the $cleaning_i$ interactions would fail to be conspiracy-resistant simply because $clean\text{–}test_i$ (eventually) always returns $'true'$, and this would obscure our example.

It is easy to see that the $get\text{–}forks_i$ interactions are still conspiracy-resistant. This property follows from the structure of process c, which is ready to interact with any fork. However, the $cleaning_i$ interactions are *not* conspiracy-resistant! To see that, consider a configuration C in which the fork-process f_i is clean (i.e., $clean_i = true$ holds), while f_{i-1} and f_{i+1} are dirty (i.e. $\neg clean_{i-1}$ and $\neg clean_{i+1}$ hold). Clearly, $c \in PA^{\mathcal{I}}_{cleaning_i}$ in C, since it readies $cleaning_i$. Let π_2 be a continuation from C, along which c is "frozen" on $cleaning_i$. Then, c will never clean any of f_{i-1} and f_{i+1}, preventing p_{i-1} and p_i from eating and f_i from getting dirty, thereby readying the $cleaning_i$ interaction. Thus, $cleaning_i$ is not conspiracy-resistant due to the failure of f_i to ready it independently of c. Hence, a hyperfair scheduler which freezes c like this could cause a deadlock (e.g., if all forks except f_i are dirty, and c is $cleaning_i\text{–}frozen$, then no interaction is enabled and the system is deadlocked).

$PHIL :: [p_1 \| \ldots \| p_n \| f_1 \| \ldots \| f_n \| c]$ where

$p_i :: \ldots$ as in Figure 1 $\ldots$

$f_i :: \ldots$ as in Figure 2 $\ldots$

$c ::$ {diligent cleaner} $c_1 := true; \ \ldots \ c_n := true;$
$*[\ \underset{i=1,n}{[]} \ c_i \wedge c_{i-1}; \ get\text{–}forks_i \, [\,] \rightarrow skip$
$\underset{i=1,n}{[]} \ give\text{–}forks_i \, [c_i := clean\text{–}test_i; \ c_{i-1} := clean\text{–}test_{i-1}] \rightarrow skip$
$\underset{i=1,n}{[]} \ \neg c_i; \ cleaning_i \, [c_i := true] \rightarrow skip$
$]$

Figure 3: Dining philosophers with a diligent cleaning service

$PHIL :: [p_1 \| \ldots \| p_n \| f_1 \| \ldots \| f_n]$ where

$p_i :: \ldots$ as in Figure 1 $\ldots$

$f_i :: l_i := true; \ r_i := true; \ c_i := true; \ sw_i := true;$
$*[\ l_i; \ get\text{–}forks_i \, [\,] \rightarrow give\text{–}forks_i \, [\,]$
$[]$
$r_i; \ get\text{–}forks_{i+1} \, [\,] \rightarrow give\text{–}forks_{i+1} [\,]$
$[]$
$switch_i \, [c_i := \neg c_i; \ sw_i := (c_i = sw_i); \ l_i := (c_i \wedge sw_i)] \rightarrow skip$
$[]$
$switch_{i+1} \, [r_i := (c_{i+1} \wedge \neg sw_{i+1})] \rightarrow skip$
$].$

Figure 4: Dining philosophers with vicious forks

Example 3 (dining philosophers with a diligent cleaner). The example in Figure 3 displays a conspiracy-resistant fork-cleaning interaction. It is not surprising, that achieving this end involves a "harder-working" cleaning process. In order to prevent the mutual-dependency of readying the *cleaning* interaction, the cleaning-process c keeps track of the cleanliness state of all forks, and readies $cleaning_i$ only if f_i is dirty. Thus, when c is $cleaning_i\text{–}frozen$, f_i is dirty and so $cleaning_i$ is enabled. Therefore, the situation in the previous example is avoided.

Example 4 (dining philosophers with vicious forks). Finally, we present (Figure 4) another variant of the dining philosophers problem in which the $get\text{–}forks_i$ interactions are *not* conspiracy-resistant and hyperfairness does not help. In this example, the neighbor forks f_i, f_{i-1} interact among themselves in a $switch_i$ interaction and thus coordinate their readiness to participate in the $get\text{–}forks_i$ interaction, i.e., the l_i boolean in f_i and the r_{i-1} boolean in f_{i-1} are never simultaneously true, and so the $get\text{–}forks_i$ interaction is never enabled. Hence conspiracy is inherent in the program semantics, and is not due to race conditions.

4 Properties of Hyperfairness

In this section we analyze hyperfairness in terms of the (briefly reviewed here) criteria of feasibility, equivalence robustness, and liveness enhancement, which are posed and formally defined in [2].

Feasibility: Any fairness assumption excludes some of the otherwise admissible computations (the "unfair" ones). A necessary property of a fairness assumption is that for every program some (finite or infinite) fair computation does exist (in other words, not all computations are excluded). Without this requirement, no scheduler could correctly treat the fairness and produce one of the fair computations. Moreover, since any reasonable scheduler cannot 'predict' the possible continuations at each point of the computation, it should be possible to extend every partial computation to a fair one.

Equivalence robustness: Concurrent systems are often modeled by means of interleaving (atomic) actions. However, the order of execution of *independent* actions in such an interleaving is arbitrary. Thus two execution sequences which are identical up to the order of independent actions are equivalent. This leads to the second criterion: a fairness assumption is *equivalence robust* if it respects this equivalence. That is, for two equivalent infinite sequences either both are fair according to the given definition, or both are unfair.

Liveness Enhancement: All models of parallel computation assume a fundamental liveness property that an action will eventually be executed in some process if the system is not deadlocked. A justification for adding an additional liveness requirement in the form of a fairness notion, is that there exists a program which has some liveness property which it would not have without the fairness notion. This criterion is termed *liveness enhancement* in order to emphasize that additional liveness properties will hold for some programs. Some fairness notions cannot force a communication to occur in a model if it did not have to occur under the fundamental liveness property. These notions are not liveness enhancing for that model.

A fairness notion is *fully adequate* iff it is feasible, equivalence-robust and liveness enhancing. The main result in this section is the establishment of the full adequacy of hyperfairness. *Feasibility* is established by presenting an *explicit scheduler* ([3]) for hyperfairness and proving its *faithfulness*. This is done below. *Liveness enhancement* follows from the examples in Section 3. *Equivalence robustness* is established by the following lemma.

Lemma: hyperfairness is equivalence robust.
Proof: Let π_1, π_2 be the interaction traces of two equivalent computations of program P. If P contains a top level interaction that is not conspiracy resistant, then every computation of P is hyperfair and we are done. Thus we now assume that every top level interaction of P is conspiracy resistant. We now establish (π_1 is hyperfair implies π_2 is hyperfair), which establishes the lemma. By definition of hyperfairness, we have that π_1 is (strongly) fair and every top level interaction is infinitely often enabled in π_1. Hence, by definition of strong fairness, every top level interaction is executed infinitely often along π_1. As π_1 and π_2 are equivalent, and therefore contain the same interaction events, every top level interaction is executed infinitely often along π_2 as well, thus π_2 is strongly fair. Also, every top level interaction must be infinitely often enabled along π_2 (in order to be infinitely often executed), hence π_2 is hyperfair. □

4.1 An explicit scheduler for hyperfairness

Let $P :: [\, P_1 \| \ldots \| P_n \,]$ be a typical IP program in normal form. Let $a_1, \ldots, a_m$, for some $m \geq 0$, be an enumeration of all the top-level interactions in P. With each interaction a_j we associate a *priority variable* z_j (as in [3]). We now define a (centralized) hyperfair scheduler H for IP. It is represented as an additional process, running in parallel with $P_i, 1 \leq i \leq n$, and having access to all of their local states, including the control positions. The priority variables z_i are local to H. We skip here the somewhat tedious task of representing the combined program in IP.

The (index of the) interaction with the highest priority is chosen by H and preserved in a local variable MIN:

$$MIN \stackrel{def.}{=} \min\{k \mid z_k = \min\{z_l \mid l = 1, \ldots, m\}\}.$$

In every process P_i, $1 \leq i \leq n$, the following modification is made: in the main (i.e., top-level) nondeterministic iteration, the local guard B_j^i (guarding an interaction body of a top-level interaction a_j in P_i) is strengthened to:

$$B_j^i \wedge (\forall \mu : i \in PA_\mu \wedge \mu = MIN \;\Rightarrow\; (j = \mu \vee \neg B_\mu^i))$$

where μ is a metavariable ranging over interaction indices. Thus, a participant of a_{MIN} does not ready any other interaction once it readies a_{MIN} (is indeed a_{MIN}-frozen), until a_{MIN} is enabled. When a_{MIN} is enabled, the following update of the z_i variables takes place. First, $z_{MIN} := ?$, (this denotes the assignment of an arbitrary positive integer to z_{MIN}) and for $k \neq MIN$, $z_k := z_k - 1$, after which MIN is redetermined, and all strengthened local guards are reevaluated. H can determine the enablement of interactions because it is centralized. In order to avoid an extra level of indexing, we assume here that each process has at most one interaction-body for each interaction a_j, $1 \leq j \leq m$.

It is important to note that the hyperfair scheduler H really acts in conjunction with a given scheduler (call it F) for the underlying fairness notion. Once H forces the enablement of an (arbitrary top-level) interaction a, (by a-freezing all the participants of a), the actual execution of a is decided by F. If F decides not to execute a, then the participants of a are unfrozen, (i.e., the priority variables are updated as described in the preceding paragraph) and execution continues. If F decides to execute a, then a is executed, after which the participants of a are unfrozen, and then execution continues. Thus, the real task of H (under the assumption that all the top-level interactions in the program P are conspiracy-resistant) is to force a situation where a is sufficiently-often enabled, and therefore eventually scheduled by F.

If some top level interactions in P are not conspiracy-resistant, then H will not guarantee the enablement of any interaction and a conspiracy may occur. In this case, the combined execution may end in a deadlock (see example 2, previous section). We assume that the conspiracy resistance of all top-level interactions is verified before the program is executed under H, and so this problem does not arise. However, in lieu of verifying the conspiracy resistance of all top-level interactions, the implementation can employ a deadlock-detection procedure, and remove all the restrictions imposed by H if deadlock ever occurs, as this would indicate that at least one top level interaction is not conspiracy resistant, and in this case, hyperfairness guarantees nothing. We now state and prove the main

theorem expressing the required properties of H.

Theorem (faithfulness of H):

For every $P \in IP$, if all top-level interactions in P are conspiracy-resistant in P, then:

(a) The restriction of every infinite fair computation of P under H (to variables of P) is a computation of P along which every top-level interaction is infinitely-often enabled (i.e., is a hyperfair computation).

(b) Every fair computation of P along which every top-level interaction is infinitely-often enabled (i.e., every hyperfair computation), can be extended (by assigning values to the $\bar{z}$ variables) to an infinite computation of P under H.

The proof (given in the appendix) is structured similarly to the ones in [3]. In proving (a), we show that once an interaction is chosen as MIN, its conspiracy-resistance guarantees its eventual enablement. Given that, it is shown that if there is a conspiracy-resistant interaction a_i not enabled from some point onwards, this means that other interactions serve as the current MIN and are repeatedly enabled, decreasing z_i until $MIN = i$ has to hold. In proving (b), the values assigned to each z_i keep track, roughly speaking, of the number of interactions that are executed until the next time a_i is enabled.

As a simple example, consider again the dining philosophers program in Figure 1. As the *get–forks* interactions have no boolean part in their guards, p_i is insistent on $get\text{–}forks_i$ once $s_i = \text{'}h\text{'}$ holds. The $get\text{–}forks_i$ interactions are conspiracy-resistant, and we have H of the form shown in Figure 5.

$$
\begin{array}{l}
H :: z_1 :=?;\ \ldots;\ z_n :=?;\ compmin; \\
\quad [\ \Box_{j=1,n}\ j = MIN \rightarrow [\ \bigwedge_{i \in PA_{get-forks_j}} ready_i(get\text{–}forks_j) \rightarrow \forall k \neq j :\ z_k := z_k - 1;\ z_j :=?;\ compmin \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad \neg \bigwedge_{i \in PA_{get-forks_j}} ready_i(get\text{–}forks_j) \rightarrow skip\ ] \\
\quad].
\end{array}
$$

Figure 5: Hyperfair scheduler for dining philosophers example

Here *compmin* is the program section

$$MIN\ :=\ \min\{k \mid z_k\ =\ \min\{z_l \mid l\ =\ 1, \ldots, n\}\}.$$

Due to the conspiracy-resistance of $get\text{–}forks_{MIN}$ and the modifications of the processes as above, $get\text{–}forks_{MIN}$ will eventually be enabled. The underlying fairness will eventually select it for execution, implying the non-starvation of p_{MIN}. Since every $get\text{–}forks_i$ interaction is eventually selected as MIN, no philosopher starves.

5 Conclusions

In this paper, a fairness notion fully-adequate for multi-party interactions, called *hyperfairness*, is presented. It guarantees that top-level *conspiracy-resistant* interactions, in which each participant infinitely-often readies the interaction independently of all other participants, are infinitely-often enabled, and by the underlying strong fairness are also infinitely-often executed. Thus hyperfairness eliminates computations where the reason for an interaction a being never enabled is simply "unfortunate" scheduling, as opposed to the case where the reason is inherent in the program semantics, in which case no fairness notion can guarantee the eventual enablement (and subsequent execution) of a.

If a program P contains some top-level interactions which are not conspiracy resistant, then, by definition, every computation of P is hyperfair, i.e., the behavior of P is unchanged by hyperfairness. Thus, as one of the referees has suggested, hyperfairness (with the underlying strong fairness) can be regarded as unconditional fairness applied to a special class of programs, namely the class of programs in which all top-level interactions are conspiracy resistant.

Though conspiracy-resistance is a rather strong property, it does occur quite often when the processes are loosely coupled in the sense that there is no strong causal dependence in sequences of consecutive interactions in which a process participates. The dining philosophers problem is a typical situation where conspiracy-resistance holds. We defer the issue of presenting a formal proof-rule for proving conspiracy-resistance to another occasion.

We believe that the study of subtle liveness properties of multi-party interaction is crucial to their understanding. A first step towards this end is proposed in this paper.

Acknowledgements: We wish to thank Shmuel Katz for useful discussions related to the definition of conspiracy-resistance and its connection to equivalence robustness. Thanks are also due to the anonymous referees, and to Leslie Lamport, who detected an error in the definition of conspiracy resistance in an earlier draft. The work started during a summer visit of the second author to the Software Technology Program of the Microelectronics and Computer Technology Corporation, and continued at the Technion, where the part of the second author was partially supported by the Fund for the Promotion of Research in the Technion, and by a grant from the Fund for Basic Research administered by the Israeli Academy of Sciences, and by MCC/STP. The first author was supported throughout by a grant from MCC/STP.

A Appendix: Proof of the faithfulness theorem.

Part a:

Follows immediately from lemma 3 established below. Let $T_h(P)$ denote the program P executed under the control of the hyperfair scheduler H.

Lemma 1:

If every top level interaction in P is conspiracy-resistant, then for every fair computation π of $T_h(P)$ and for every configuration C along π: if $a_{MIN} = a$ in C, then along the continuation of π from C interaction a is eventually enabled.

Proof:

Let π' be the suffix of π starting in configuration C. The proof is by induction over the set of processes $PA_a^{\mathcal{I}}$ that are insistent on a. When $a_{MIN} = a$, the behavior of $T_h(P)$ and the $(a, PA_a^{\mathcal{I}})$-derived program $P_{a,PA_a^{\mathcal{I}}}$ are identical, since in both cases, every process that readies a is subsequently a-frozen until a is enabled, and processes that do not ready a are not a-frozen. Hence, π' is also a (suffix of a) computation of $P_{a,PA_a^{\mathcal{I}}}$, and so, by definition of conspiracy resistance, there exists a participant $P_j \in (PA_a - PA_a^{\mathcal{I}})$ such that P_j eventually readies a along π', and hence joins $PA_a^{\mathcal{I}}$. Thus, by induction over $PA_a^{\mathcal{I}}$, we conclude that, eventually, $PA_a^{\mathcal{I}} = PA_a$ holds along π', i.e., a is enabled. □

Lemma 2:

If every top level interaction in P is conspiracy-resistant, then $(\bigwedge_{i=1,n} z_i > -n)$ is an invariant of $T_h(P)$, where n is the total number of top-level interactions.

Proof:

Let z_k be an arbitrary priority variable. It suffices to show that $z_k > -n$ is invariant. We assume that all z_i variables are initially nonnegative, and so $z_k > -n$ holds initially.

Assume that $z_k = 0$ and $MIN = j$ hold before a priority variable update (for some $j \neq k$). Thus both $z_k := z_k - 1$ and $z_j := ?$ are executed (see section 4.1 for a description of the hyperfair scheduler H). Hence $z_j > z_k$ holds after the update, and it is obvious that $z_j > z_k$ will continue to hold until $z_k := ?$ is executed, since each priority variable update prior to the execution of $z_k := ?$ simply decrements each of z_j, z_k by one. Therefore a_j cannot be the MIN interaction until $z_k := ?$ is executed, i.e., until after a_k has become the MIN interaction. Since j is arbitrary, we conclude that each of the $(n-1)$ interactions other than a_k can be the MIN interaction at most once until after a_k has become the MIN interaction, and so at most $n-1$ priority variable updates can take place until $z_k := ?$ is executed, so z_k can be decremented at most $n-1$ times before being assigned an arbitrary positive value.

Now if $z_k > 0$ holds, then, since z_k is decremented by one, it follows that $z_k = 0$ will hold before z_k becomes negative, and the argument in the previous paragraph also applies in this case. Since π is an arbitrary computation, we conclude that $z_k > -n$ is invariant, and the lemma is established. □

Lemma 3: If every top-level interaction of program P is conspiracy-resistant, then every top-level interaction is enabled infinitely-often along any infinite fair computation of $T_h(P)$.

Proof:

Assume otherwise, i.e, there exist an infinite fair computation π of $T_h(P)$ and a top-level interaction a_k of P such that a_k is enabled only finitely often along π. Thus there is a suffix π' of π such that a_k is never enabled along π'. Now consider an arbitrary configuration C of π' . By definition of MIN, we have that $MIN = j$ holds in C for some interaction a_j. Thus by lemma 1, a_j is eventually enabled, say at a configuration C' that follows configuration C along π'. By construction of the hyperfair scheduler H, the priority variables will be updated in configuration C', and so z_k will be decremented. Since C is an arbitrary configuration, it is clear that the above argument can be repeated for every configuration of π', hence we conclude that z_k is decremented infinitely often along π'. Since a_k is never enabled along π', $z_k := ?$ is never executed along π', and z_k is decremented infinitely often along π', this means that eventually, $z_k < -n$ holds. But this contradicts lemma 2, and so the initial assumption is false and there cannot exist such a computation π and interaction a_k. □

Part b:

Let π be a fair computation of P, along which every top-level interaction is infinitely-often enabled. For any state σ_j along π, and any top-level interaction a_l, define:

$$k_{l,j} \;=\; \min\{i \mid i \geq j \text{ and } a_l \text{ is enabled in } \sigma_i\}$$

Extending to σ'_j (a state of $T_h(P)$) is done inductively. First, $\sigma'_0(z_l) = k_{l,0}$. Let $\sigma_j(MIN) = m$, then:

$$\sigma'_{j+1}(z_l) \;=\; \begin{cases} \sigma'_j(z_l) & \text{if } a_m \text{ is not enabled in } \sigma_j \\ \sigma'_j(z_l) - 1 & \text{if } a_m \text{ is enabled in } \sigma_j \text{ and } l \neq m \\ k_{l,j+1} & \text{if } a_m \text{ is enabled in } \sigma_j \text{ and } l = m \end{cases}$$

The computation π' obtained this way is a computation of $T_h(P)$. The interaction a_{MIN} is always eventually enabled along π', whereupon z_{MIN} is reset to a natural number, and for $l \neq MIN$, z_l is decreased by 1.

References

[1] K.R. Apt, L. Bouge, Ph. Clermont: "Two normal form theorems for CSP programs", IPL 26, 1987, pp. 165-171.

[2] K.R. Apt, N. Francez, S. Katz: "Appraising Fairness in Distributed Languages", Distributed Computing 2:226-241, August 1988. Also: proc. 14th ACM-POPL symposium, Munich, Germany, Jan. 1987.

[3] K.R. Apt, E.-R. Olderog: "Proof Rules and Transformations Dealing With Fairness", Science of Computer Programming 3, pp. 65-100, 1983.

[4] R. Bagrodia: "A Distributed Algorithm to Implement N-Party Rendezvous", TR, Dept. of Computer Science, Univ. of Texas at Austin, June 1987.

[5] R.J.R. Back, R. Kurki-Suonio: "Decentralization of Process Nets With Centralized Control", Distributed Computing 3:73-87, 1989. Also: proc. 2nd ACM-PODC, Montreal, Canada, August 1983.

[6] R.J.R. Back, R. Kurki-Suonio: "Cooperation in distributed systems using symmetric multiprocess handshaking", TR A34, Abo Akademi, 1984.

[7] R.J.R. Back, R. Kurki-Suonio: "Serializability in Distributed Systems With Handshaking", TR 85-109, CMU, 1985.

[8] R.J.R. Back, R. Kurki-Suonio: "Distributed cooperation with action systems", ACM-TOPLAS 10,4: 513-554, October 1988

[9] A. Charlesworth: "The Multiway Rendezvous", ACM-TOPLAS 9, 2: 350-366, July 1987.

[10] K.M. Chandy, J. Misra: "Synchronizing Asynchronous Processes - the Committee-Coordination Problem", TR, Dept. of Computer Science, Univ. of Texas at Austin, 1987.

[11] K.M. Chandy, J. Misra: **Parallel Program Design: A Foundation**, (chapter 14), Addison Wesley, 1988.

[12] E.W. Dijkstra: **A Discipline of Programming**, Prentice-Hall, 1976.

[13] N. Francez, W.P. de Roever: "Fairness in Communicating Processes", unpublished memo, Computer Science Dept., Utrecht University, July 1980.

[14] N. Francez, B.T. Hailpern, G. Taubenfeld: "SCRIPT - A Communication Abstraction Mechanism and Its Verification", Science of Computer Programming 6,1, pp. 35-88, Jan. 1986.

[15] I.R. Forman: "On the Design of Large Distributed Systems", TR STP-098-86 (Rev. 1.0), MCC, Austin, TX, Jan. 1987. A preliminary version presented at the First International Conf. on Computer Languages, Miami, FL, October 1986.

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

[17] N. Francez: "Cooperating proofs for distributed programs with multi-party interactions", IPL Vol. 32, No. 5, pp. 235-242, September 1989.

[18] O. Grumberg, N. Francez, S. Katz: "Fair Termination of Communicating Processes", 3rd ACM-PODC Conference, Vancouver, BC, Canada, August 1984.

[19] C.A.R. Hoare: "Communicating Sequential Processes", CACM 21,8, pp. 666-678, August 1978.

[20] C.A.R. Hoare: **Communicating Sequential Processes**, Prentice-Hall, 1985.

[21] R. Kuiper, W.P. de Roever: "Fairness Assumptions for CSP in a Temporal Logic Framework", proc. TC.2 Working Conference on Formal Description of Programming Concepts, Garmisch Partenkirchen (D. Biorner, ed.), North Holland, 1983.

[22] E.-R. Olderog, K.R. Apt: "Transformations Realizing Fairness Assumptions for Parallel Programs", ACM-TOPLAS 10,3: 420-455, July 1988.

[23] G.D. Plotkin: "An Operational Semantics for CSP", TC.2 working group conference on the formal description of programming concepts, Garmisch Partenkirchen, (D. Biorner, ed.), North Holland, 1983.

[24] A. Pnueli: lecture notes of CS395T, "Specification and Verification of Reactive Systems", Univ. of Texas at Austin, fall 1986 (notes taken by Charlie Richter).

[25] S. Ramesh, H. Mehndiratta: "A methodology for developing distributed programs", IEEE-TSE vol. SE-13, 8: 967-976, August 1987.