

# Fairness and Hyperfairness in Multi-party Interactions <sup>†</sup>

(Extended Abstract)

Paul C. Attie<sup>1</sup>

Nissim Francez<sup>2,3</sup>

Orna Grumberg<sup>3</sup>

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

## 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 [AFK88]. 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. Fairness properties at the language level (e.g., an action that is sufficiently often enabled is eventually executed) are used to verify that programs written in that language satisfy their progress specification.

---

<sup>†</sup>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 also by MCC/STP. The first author was supported throughout by a grant from MCC/STP.

In this paper, we are interested in formulating a fairness notion appropriate for concurrent programs expressed in programming languages with *multi-party interactions* as their primitive construct for interprocess synchronization and communication. 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 (in more than one way) 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* [FHT86], *joint actions* [BKS83], *shared actions* [RM87], *compacts* [Ch87] and *teams and interactions* [Fo87]. In addition, the *handshakes* in Hoare's algebraic version of *CSP* [Ho85] 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 [BKS84, BKS88, Ba87, CM87, CM88].

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) [GFK84, BKS84, BKS85, AFK88, Fr86], 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.

A *conspiracy* occurs if, from a certain stage in a computation, an interaction is never enabled for execution because conflicting interactions intermittently engage some of the needed participants. This situation occurs in both serialized and overlapping semantics. What is needed is a fairness notion that prevents this behavior whenever it is due solely to “race conditions”, i.e., whenever its prevention involves delaying some interactions and proceeding with independent, non-conflicting interactions (this is made more precise in the sequel). In this paper, we formulate such a fairness notion for multi-party interactions. Our notion also satisfies the three semantic criteria posed in [AFK88] 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 (to our knowledge). The existence of such a fairness notion was presented as an open problem by Pnueli [Pn86]. 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. Section 2 presents a mini-language, an abstraction of Radle [Fo87], in terms of which *hyperfairness*, the fairness

notion proposed for multi-party interactions, is defined. Both its serialized and overlapping semantics are given. In Section 3 hyperfairness is defined, and examples are presented, which rely on hyperfairness for their required liveness properties. In Section 4 an *explicit scheduler* transformation [AO83, OA88] for hyperfairness is proposed and proven to be *faithful*, i.e., it generates all and only hyperfair computations of a given program. Section 5 presents our conclusions.

## 2 The Language IP (Interacting Processes)

In this section we present a simple mini-language, called *IP (Interacting Processes)*, first introduced in [Fr89]. 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 [Dij76], which has only boolean guards, and *CSP* [Ho78], 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 [Fr89].

A program  $P :: [P_1 || \dots || P_n]$  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  $S_1; \dots; S_k$  for any  $k \geq 2$ .

**nondeterministic selection** -

$\big[ \big[_{k=1,m} B_k; a_k [\bar{v}_k := \bar{e}_k] \rightarrow S_k \big]$ : 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** -

$* \big[ \big[_{k=1,m} B_k; a_k [\bar{v}_k := \bar{e}_k] \rightarrow S_k \big]$ : Similar to the nondeterministic selection, but execution terminates once no open guards exist, and execution of the whole state-

ment 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 [Pl83]. We define two different semantics: *serialized* and *overlapping* (compare with a similar distinction in [BKS85] and [GFK84]). 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*  $\langle [S_1 || \dots || S_n], \sigma \rangle$  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  $\sigma$  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  $\langle [E || \dots || E], \sigma \rangle$  is a *terminal* configuration. For a state  $\sigma$ , we use the usual notions of a *variant*  $\sigma[c/x]$  and  $\sigma[\bar{c}/\bar{x}]$ , obtained from  $\sigma$  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  $\sigma$ . 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.

$$\begin{aligned} & \langle [S_1 \parallel \dots \parallel S_i \parallel \dots \parallel S_n], \sigma \rangle \rightarrow \\ & \quad \langle [S_1 \parallel \dots \parallel E \parallel \dots \parallel S_n], \sigma \rangle \end{aligned} \quad (1)$$

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

$$\begin{aligned} & \langle [S_1 \parallel \dots \parallel S_i \parallel \dots \parallel S_n], \sigma \rangle \rightarrow \\ & \quad \langle [S_1 \parallel \dots \parallel E \parallel \dots \parallel S_n], \sigma[\sigma(e)/x] \rangle \end{aligned} \quad (2)$$

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

$$\begin{aligned} & \langle [S_1 \parallel \dots \parallel S_{i_1-1} \parallel S_{i_1} \parallel S_{i_1+1} \parallel \dots \parallel S_{i_k-1} \parallel S_{i_k} \parallel S_{i_k+1} \parallel \dots \parallel S_n], \sigma \rangle \\ & \rightarrow \\ & \langle [S_1 \parallel \dots \parallel S_{i_1-1} \parallel S'_{i_1} \parallel S_{i_1+1} \parallel \dots \parallel S_{i_k-1} \parallel S'_{i_k} \parallel S_{i_k+1} \parallel \dots \parallel S_n], \sigma' \rangle \end{aligned} \quad (3)$$

iff the following holds: There is an interaction  $a$  with a set of participants  $PA_a = \{i_1, \dots, i_k\}$  (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)  $S_i = [\bigcup_{j=1, n_i} B_j; a_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$  and there exists some  $j$ ,  $1 \leq j \leq n_i$ , s.t.  $B_j$  holds in  $\sigma$ ,  $a_j = a$  and  $S'_i = T_j$
- (c)  $S_i = *[\bigcup_{j=1, n_i} B_j; a_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$  and there exists some  $j$ ,  $1 \leq j \leq n_i$ , s.t.  $B_j$  holds in  $\sigma$ ,  $a_j = a$ ,  $S'_i = T_j$ ;  $S_i$ .

Finally, for all these cases,  $\sigma' = \sigma[\sigma(\bar{e})/\bar{v}]$ , with  $\bar{v} = \bigcup_{i \in PA_a} \bar{v}_i$ ,  $\bar{e} = \bigcup_{i \in PA_a} \bar{e}_i$ , where  $\cup$  denotes a union operation which is ordered by process index.

For any  $1 \leq i \leq n$ , if

$$\begin{aligned} & \langle [S_1 \parallel \dots \parallel S_i \parallel \dots \parallel S_n], \sigma \rangle \rightarrow \\ & \quad \langle [S'_1 \parallel \dots \parallel S'_i \parallel \dots \parallel S'_n], \sigma' \rangle \\ & \text{then} \\ & \langle [S_1; T_1 \parallel \dots \parallel S_i; T_i \parallel \dots \parallel S_n; T_n], \sigma \rangle \rightarrow \\ & \quad \langle [S'_1; T_1 \parallel \dots \parallel S'_i; T_i \parallel \dots \parallel S'_n; T_n], \sigma' \rangle \end{aligned} \quad (4)$$

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*  $\pi$  of  $P$  on  $\sigma$  is a maximal (finite or infinite) sequence of configurations  $C_i$ ,  $i \geq 0$ , such that:

(a)  $C_0 = \langle P, \sigma \rangle$ .

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

(2) The computation  $\pi$  *terminates* iff it is finite and its last configuration is terminal;  $\pi$  *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 set of participants, i.e.,  $PA_{a_1} \cap PA_{a_2} \neq \emptyset$ .

(5) Two interactions  $a_1$  and  $a_2$  are *independent* iff  $PA_{a_1} \cap PA_{a_2} = \emptyset$ , i.e., they have no common participant.

(6) Two computations are equivalent iff they differ only in the order in which independent interactions are executed.

(7) 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  $\rho$ . If  $\rho[i] = \text{true}$ ,  $1 \leq i \leq n$ , 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  
 $\langle [S_1 || \dots || S_n], \sigma, (\rho[1], \dots, \rho[i-1], \text{false}, \rho[i+1], \dots, \rho[n]) \rangle$   
 $---\rightarrow$   
 $\langle [S_1 || \dots || S_n], \sigma, (\rho[1], \dots, \rho[i-1], \text{true}, \rho[i+1], \dots, \rho[n]) \rangle$   
 for every  $1 \leq i \leq n$ .

An *overlapping* computation  $\pi$  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  $\rho[i] = \text{true}$  for all  $1 \leq i \leq n$ .

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

(c) In an infinite computation,  $\rho[i] = \text{true}$  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  $\rho[i] = \text{true}$ . We denote this property by  $\text{ready}_i(a)$ . The interaction  $a$  is *enabled* in a configuration  $C$  iff it is jointly readied in  $C$  by all its participants, i.e.,  $\bigwedge_{i \in PA_a} \text{ready}_i(a)$  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 [ABC87] 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.

## 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 [FdR80, KdR83, BKS85, GFK84] (see also [Fr86], chap-

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

ter 5), the classification of fairness notions for multi-party interactions presented in [AFK88] 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 enabledness conditions required to guarantee eventual execution of the subject action. See [Fr86] 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.

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

As noted in [AFK88], 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  $PA_a^i \cup PA_a^p$ , such that all members of  $PA_a^i$  are continuously ready to participate in the interaction  $a$ , while members of  $PA_a^p$  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 table 5), 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 table 2) 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  $PA_a^p$  can infinitely often ready  $a$  *independently* of  $PA_a^i$  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 enabledness condition of this “underlying” fairness notion is met. Since members of  $PA_a^p$  ready  $a$  independently of  $PA_a^i$  and of each other, they can be successively “frozen” in a state in which they ready  $a$  (thereby joining  $PA_a^i$ ), 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  $\pi$  iff it continuously readies  $a$  in  $\pi$  from that point onwards.
- (2) A process  $P_i$  is *persistent* on an interaction  $a$  in a computation  $\pi$  iff it readies  $a$  infinitely-often in  $\pi$ .

**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  $\pi$  the following condition holds:

Let  $\pi_1$  be a finite prefix of  $\pi$  with final configuration  $C = \langle S, \sigma \rangle$ , and let  $PA_a^i$  be the set of all the participants of  $a$  that ready  $a$  in  $C$ . Then, for every fair computation  $\pi_2$  of the  $(a, PA_a^i)$ -derived program  $S_{a, PA_a^i}$  (starting in state  $\sigma$ ), there exists a participant  $P_j \in (PA_a - PA_a^i)$  such that  $P_j$  eventually readies  $a$  along  $\pi_2$ .

We refer to this as each process in  $PA_a^i$  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 enabledness of  $a$ . Note that in the above definition  $PA_a^i = \emptyset$  is possible. We now come to the definition of our central notion.

**Definition** (Hyperfairness).

Let  $P$  be an *IP* program in which every top-level interaction is conspiracy-resistant. Then, an infinite computation  $\pi$  of  $P$  is *hyperfair* iff the following condition is met:

$\pi$  is fair and every top-level interaction is infinitely-often enabled in  $\pi$ . Furthermore, 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  $\pi$  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 enabledness 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 ( $\pi$ ) of  $P$  but not in others, i.e., it is a function of  $a$ ,  $P$ , and  $\pi$ , whereas conspiracy resistance is not a 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 enabledness condition, e.g., if  $a$  is never enabled along a computation  $\pi$ , then  $\pi$  is vacuously fair with respect to  $a$ . However, this is not the case with hyperfairness. If all top level interactions are conspiracy resistant, then  $\pi$  is not hyperfair with respect to interaction  $a$  *because*  $a$  is not enabled at any point of  $\pi$ . 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 enabledness conditions for weak interaction fairness. We now present several examples which illustrate hyperfairness.

---

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

$$\begin{aligned}
p_i &:: s_i := 't'; \\
&*[ s_i := 't' \rightarrow s_i := 'h' \\
&\quad \square \\
&\quad s_i := 'h'; \text{get-forks}_i[s_i := 'e'] \rightarrow \text{give-forks}_i[s_i := 't'] \\
&\quad ], \\
f_i &:: *[ \text{get-forks}_i[] \rightarrow \text{give-forks}_i[] \\
&\quad \square \\
&\quad \text{get-forks}_{i+1}[] \rightarrow \text{give-forks}_{i+1}[] \\
&\quad ].
\end{aligned}$$


---

Table 2: A starvation free solution of the dining philosophers problem in hyperfair **IP**

**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 Table 2. 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  $\text{get-forks}_i$  (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  $\text{give-forks}_i$  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  $\text{get-forks}_2$  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  $\pi$  of a serialized computation

$$\pi = \text{get-forks}_1 \text{get-forks}_3 (\text{give-forks}_1 \text{get-forks}_1 \text{give-forks}_3 \text{get-forks}_3)^\omega$$

It is clear that  $\text{get-forks}_2$  is never enabled in  $\pi$  because  $\text{get-forks}_1$  and  $\text{get-forks}_3$  alternately engage  $f_1$  and  $f_2$  respectively. Note that the conspiracy against  $\text{get-forks}_2$  in  $\pi$  is due to “unfortunate” scheduling which prevents  $f_1$  and  $f_2$  from readying  $\text{get-forks}_2$  simultaneously. This is an example of conspiracy occurring in the serialized semantics, since  $\pi$  is a serialized computation. Now  $p_i$  is insistent on  $\text{get-forks}_i$  (once  $s_i = 'h'$ ), and the two fork processes  $f_i$  and  $f_{i-1}$  ready

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

$$\pi' = \text{get-forks}_1 \text{get-forks}_3 (\text{give-forks}_1 \text{give-forks}_3 \text{get-forks}_1 \text{get-forks}_3)^\omega$$

in which the  $\text{get-forks}_1$  interaction is commuted with the independent  $\text{give-forks}_3$  interaction, no conspiracies occur, and  $\text{get-forks}_2$  is infinitely often enabled (it is enabled immediately after  $\text{get-forks}_3$  has been executed). But  $\pi'$  is also not hyperfair, because  $\text{get-forks}_2$  is infinitely often enabled but never executed, and so  $\pi'$  is not fair.

**Example 2** (dining philosophers with fork cleaning). In the previous example, the two forks readied the  $\text{get-forks}_i$  interaction unconditionally. Here is a somewhat strengthened example (Table 3), 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  $\text{clean-test}_i$  represents some hidden condition, returning a boolean value representing the state of cleanliness of a fork. It is easy to see that the  $\text{get-forks}_i$  interactions are still conspiracy-resistant. This property follows from the structure of process  $c$ , which is ready to interact with any fork. However, the  $\text{cleaning}_i$  interactions are *not* conspiracy-resistant! To see that, consider a configuration  $C$  in



---

$PHIL :: [p_1 \parallel \dots \parallel p_n \parallel f_1 \parallel \dots \parallel f_n \parallel c]$  where

$p_i :: \dots$  as before  $\dots$

$f_i :: clean_i := true;$   
 $\quad * [clean_i; get-forks_i [] \rightarrow give-forks_i [clean_i := clean-test_i]$   
 $\quad \parallel$   
 $\quad clean_i; get-forks_{i+1} [] \rightarrow give-forks_{i+1} [clean_i := clean-test_i]$   
 $\quad \parallel$   
 $\quad \neg clean_i; cleaning_i [clean_i := true] \rightarrow skip$   
 $\quad ],$   
 $c :: \{cleaning\ service\} * [\bigparallel_{i=1,n} cleaning_i [] \rightarrow skip]$

---

Table 3: Dining philosophers with a cleaning service

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,  $c \in PA^i_{cleaning_i}^\dagger$  in  $C$ , since it readies  $cleaning_i$ . Let  $\pi_2$  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  $cleaning_i$ -frozen, then no interaction is enabled and the system is deadlocked).

**Example 3** (dining philosophers with a diligent cleaner). The example in Table 4 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  $cleaning_i$ -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 (Table 5) another variant of the dining philosophers problem in which the  $get-forks_i$

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  $get-forks_i$  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  $get-forks_i$  interaction is never enabled. Hence conspiracy is inherent in the program semantics, and is not due to race conditions. The program for  $p_i$  is as in Table 2.

## 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 [AFK88].

**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

---

<sup>†</sup>note that the superscript  $i$  denotes “insistence” while the sub-subscript  $i$  is an index over  $1 \dots n$

---

$PHIL :: [p_1 \parallel \dots \parallel p_n \parallel f_1 \parallel \dots \parallel f_n \parallel c]$  where

$p_i :: \dots$  as before  $\dots$

$f_i :: \dots$  as in Table 3  $\dots$

$c :: \{\text{diligent cleaner}\} \ c_1 := \text{true}; \dots \ c_n := \text{true};$   
 $\quad * [ \bigparallel_{i=1,n} \ c_i \wedge c_{i-1}; \text{get-forks}_i [] \rightarrow \text{skip}$   
 $\quad \quad \bigparallel_{i=1,n} \ \text{give-forks}_i [c_i := \text{clean-test}_i; c_{i-1} := \text{clean-test}_{i-1}] \rightarrow \text{skip}$   
 $\quad \quad \bigparallel_{i=1,n} \ \neg c_i; \text{cleaning}_i [c_i := \text{true}] \rightarrow \text{skip} ]$

---

Table 4: Dining philosophers with a diligent cleaning service

---

$PHIL :: [p_1 \parallel \dots \parallel p_n \parallel f_1 \parallel \dots \parallel f_n]$  where

$p_i :: \dots$  as before  $\dots$

$f_i :: l_i := \text{true}; r_i := \text{true}; c_i := \text{true}; sw_i := \text{true};$   
 $\quad * [ l_i; \text{get-forks}_i [] \rightarrow \text{give-forks}_i []$   
 $\quad \quad \bigparallel$   
 $\quad \quad r_i; \text{get-forks}_{i+1} [] \rightarrow \text{give-forks}_{i+1} []$   
 $\quad \quad \bigparallel$   
 $\quad \quad \text{switch}_i [c_i := \neg c_i; sw_i := (c_i = sw_i); l_i := (c_i \wedge sw_i)] \rightarrow \text{skip}$   
 $\quad \quad \bigparallel$   
 $\quad \quad \text{switch}_{i+1} [r_i := (c_{i+1} \wedge \neg sw_{i+1})] \rightarrow \text{skip}$   
 $\quad \quad ]$ .

---

Table 5: Dining philosophers with vicious forks

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* ([AO83]) 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  $\pi_1, \pi_2$  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 ( $\pi_1$  is hyperfair implies  $\pi_2$  is hyperfair), which establishes the lemma. By definition of hyperfairness, we have that  $\pi_1$  is (strongly) fair and every top level interaction is in-

finitely often enabled in  $\pi_1$ . Hence, by definition of strong fairness, every top level interaction is executed infinitely often along  $\pi_1$ . As  $\pi_1$  and  $\pi_2$  are equivalent, and therefore contain the same interaction events, every top level interaction is executed infinitely often along  $\pi_2$  as well, thus  $\pi_2$  is strongly fair. Also, every top level interaction must be infinitely often enabled along  $\pi_2$  (in order to be infinitely often executed), hence  $\pi_2$  is hyperfair.  $\square$

## 4.1 An explicit scheduler for hyperfairness

Let  $P$  be a typical  $IP$  program in normal form. Let  $a_1, \dots, a_m$ , 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 [AO83]). We now define a 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  $MIN$ :

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

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

$$B_j^i \wedge (i \in PA_{MIN} \Rightarrow (j = MIN \vee \neg B_{MIN}^i)).$$

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. In order to avoid an extra level of indexing, we assume here that each process has at most one interaction-body for each interaction  $a_j, 1 \leq j \leq m$ .

---


$$\begin{aligned}
H :: & z_i := ?, \dots, z_n := ?; \text{compmin}; \\
& [ \bigwedge_{j=1, n} j = MIN \rightarrow [ \bigwedge_{i \in P_{A_{\text{get-forks}_j}}} \text{ready}_i(\text{get-forks}_j) \rightarrow \forall k \neq j : z_k := z_k - 1; z_j := ?; \text{compmin} \\
& \quad \bigwedge \\
& \quad \neg \bigwedge_{i \in P_{A_{\text{get-forks}_j}}} \text{ready}_i(\text{get-forks}_j) \rightarrow \text{skip} ] \\
& ].
\end{aligned}$$


---

Table 6: Hyperfair scheduler for dining philosophers example

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. The real task of  $H$  (under the assumption that all the interactions in the program  $P$  are conspiracy-resistant) is to force a situation where each interaction is sufficiently-often enabled, and therefore eventually scheduled by  $F$ . However, if some top level interactions in  $P$  are not conspiracy-resistant,  $H$  will not guarantee the enabledness of any interaction and a conspiracy may occur. In this case, the combined execution may end in a deadlock (see example 2, previous section). Thus, the implementation should 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  $z$  variables) to an infinite computation of  $P$  under  $H$ .

The proof (given in the full paper) is structured similarly to the ones in [AO83]. In proving (a), we

show that once an interaction is chosen as  $MIN$ , its conspiracy-resistance guarantees its eventual enabledness. 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 Table 2. As the *get-forks* interactions have no boolean part in their guards,  $p_i$  is insistent on *get-forks* <sub>$i$</sub>  once  $s_i = 'h'$  holds. The *get-forks* <sub>$i$</sub>  interactions are conspiracy-resistant, and we have  $H$  of the form shown in table 6. Here *compmin* is the program section

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

Due to the conspiracy-resistance of *get-forks* <sub>$MIN$</sub>  and the modifications of the processes as above, *get-forks* <sub>$MIN$</sub>  will eventually be enabled. The underlying fairness will eventually select it for execution, implying the non-starvation of  $p_{MIN}$ . Since every *get-forks* <sub>$i$</sub>  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 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  $a$ .

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 Leslie Lamport who detected an error in the definition of conspiracy resistance in an earlier draft.

## References

- [ABC87] K.R. Apt, L. Bouge, Ph. Clermont: “Two normal form theorems for CSP programs”, IPL 26, 1987, pp. 165-171.
- [AFK88] 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.
- [AO83] K.R. Apt, E.-R. Olderog: “Proof Rules and Transformations Dealing With Fairness”, Science of Computer Programming 3, pp. 65-100, 1983.
- [Ba87] R. Bagrodia: “A Distributed Algorithm to Implement N-Party Rendezvous”, TR, Dept. of Computer Science, Univ. of Texas at Austin, June 1987.
- [BKS83] 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.
- [BKS84] R.J.R. Back, R. Kurki-Suonio: “Cooperation in distributed systems using symmetric multiprocess handshaking”, TR A34, Abo Akademi, 1984.
- [BKS85] R.J.R. Back, R. Kurki-Suonio: “Serializability in Distributed Systems With Handshaking”, TR 85-109, CMU, 1985.
- [BKS88] R.J.R. Back, R. Kurki-Suonio: “Distributed cooperation with action systems”, ACM-TOPLAS 10,4: 513-554, October 1988
- [Ch87] A. Charlesworth: “The Multiway Rendezvous”, ACM-TOPLAS 9, 2: 350-366, July 1987.
- [CM87] K.M. Chandy, J. Misra: “Synchronizing Asynchronous Processes - the Committee-Coordination Problem”, TR, Dept. of Computer Science, Univ. of Texas at Austin, 1987.
- [CM88] K.M. Chandy, J. Misra: **Parallel Program Design: A Foundation**, (chapter 14), Addison Wesley, 1988.
- [Dij76] E.W. Dijkstra: **A Discipline of Programming**, Prentice-Hall, 1976.
- [FdR80] N. Francez, W.P. de Roever: “Fairness in Communicating Processes”, unpublished memo, Computer Science Dept., Utrecht University, July 1980.

- [FHT86] 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.
- [Fo87] 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.
- [Fr86] N. Francez: **Fairness**, Springer-Verlag, 1986.
- [Fr89] N. Francez: "Cooperating proofs for distributed programs with multi-party interactions", *IPL Vol. 32, No. 5*, pp. 235-242, September 1989.
- [GFK84] O. Grumberg, N. Francez, S. Katz: "Fair Termination of Communicating Processes", 3rd ACM-PODC Conference, Vancouver, BC, Canada, August 1984.
- [Ho78] C.A.R. Hoare: "Communicating Sequential Processes", *CACM* 21,8, pp. 666-678, August 1978.
- [Ho85] C.A.R. Hoare: **Communicating Sequential Processes**, Prentice-Hall, 1985.
- [KdR83] 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.
- [OA88] E.-R. Olderog, K.R. Apt: "Transformations Realizing Fairness Assumptions for Parallel Programs", *ACM-TOPLAS* 10,3: 420-455, July 1988.
- [Pl83] 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.
- [Pn86] A. Pnueli: lecture notes of CS395T, "Specification and Verification of Reactive Systems", Univ. of Texas at Austin, fall 1986 (notes taken by Charlie Richter).
- [RM87] S. Ramesh, H. Mehndiratta: "A methodology for developing distributed programs", *IEEE-TSE vol. SE-13, 8*: 967-976, August 1987.