Fairness in systems based on multiparty interactions

David Ruiz*,†, Rafael Corchuelo and Miguel Toro

ETSI Informática, Avda. de la Reina Mercedes s/n, Sevilla E-41012, Spain

SUMMARY

In the context of the Multiparty Interaction Model, *fairness* is used to insure that an interaction that is enabled sufficiently often in a concurrent program will eventually be selected for execution. Unfortunately, this notion does not take *conspiracies* into account, i.e. situations in which an interaction never becomes enabled because of an unfortunate interleaving of independent actions; furthermore, *eventual execution* is usually too weak for practical purposes since this concept can only be used in the context of infinite executions. In this article, we present a new fairness notion, *k-conspiracy-free fairness*, that improves on others because it takes finite executions into account, alleviates conspiracies that are not inherent to a program, and *k* may be set a priori to control its goodness to address the above-mentioned problems.

KEY WORDS: concurrent programs; multiparty interactions; fairness; fair finiteness; conspiracies

1. INTRODUCTION

In this article, we focus on fairness in concurrent systems that use the Multiparty Interaction ($\mathcal{MI}$) model. A multiparty interaction is an abstraction that allows several processes to synchronize and exchange information coordinately. Fairness becomes essential in $\mathcal{MI}$-based systems since an $\mathcal{MI}$-based process may offer to participate in several interactions, although it can execute only one at a time [1]. Intuitively, an execution of a program is fair iff every interaction that is ready for execution sufficiently often is executed sufficiently often, which avoids executions in which such interactions are neglected.

Notice that '*sufficiently often*' is a vague term, which implies that there is not a single prevailing definition. However, many researchers agree in that so-called *strong fairness* deserves attention

*Correspondence to: David Ruiz, ETSI Informática, Avda. de la Reina Mercedes s/n, Sevilla E-41012, Spain.
†E-mail: druiz@tdg.lsi.us.es

Contract/grant sponsor: Spanish Ministry of Science and Technology; contract/grant numbers: TIC-2000-1106-C02-01; FIT-150100-2001-78; TAMANSI PCB-02-001

because it may induce desirable properties such as termination or eventual response to a request for service [1–3]. Unfortunately, this notion is not restrictive enough since it does not take finite executions into account, and an interaction might never get ready for execution because of an unfortunate interleaving of independent actions that might prevent some of the processes that need it to coordinate from engaging it at the right time. This has motivated several authors to work on stronger fairness notions, but none of them solves both problems simultaneously [4–7].

The main contribution we present in this article is a new fairness notion called *k-conspiracy-free fairness* that addresses the above-mentioned problems by fine-tuning the value we assign to k. We also present a framework we have devised to implement fair $\mathcal{MI}$-based systems and report on the results of an experimental analysis we conducted to compare our proposal with others. From these results, we conclude that our proposal allows one to control conspiracies efficiently and it is easier to apply since our implementation is generic, i.e. it is not a transformational approach and thus needs not transform the source code of the systems to which it is applied.

The rest of the article is organized as follows. In Section 2, we report on some related work about multiparty interactions, fairness notions, and summarize how we improve other authors' work. The foundations of our framework are presented in Section 3 and in Section 4, we introduce our fairness notion and define it rigorously. We report on how to implement our fairness notion and on our experimental results in Sections 5 and 6, respectively. Finally, some conclusions are drawn from previous parts and summarized in Section 7.

2. RELATED WORK

In this section, we first present the $\mathcal{MI}$ model. Later, we report on current fairness proposals and argue on their deficiencies. Finally we summarize our contributions.

2.1. Multiparty interactions

The $\mathcal{MI}$ model provides interactions as the sole means for process synchronization and communication [1,8–10], and it has been proven to achieve optimal concurrency and/or parallelism in some common situations [11].

Contrarily to the usual message-passing model, which emphasizes two processes exchanging messages and, thus, communication, multiparty interactions focus on agreement amongst multiple parties that need to cooperate in order to achieve a common goal, e.g. transferring money from a bank to another by means of a point of sales terminal (three processes) [12], paying taxes on-line (three processes in Spain: a taxpayer, the Exchequer, and Spain's Certification Authority), filtering in e-commerce [13] (a customer, a filter system, and several service providers), or reaching a virtual agreement in an auction sale (multiple processes). Reference [10] provides a complete taxonomy of languages that support this interaction model, and recent contributions presented in [12,14,15] have extended Java to support multiparty interactions to some extent. In [8], the model was further researched and combined with aspect orientation.

Roughly speaking, a multiparty interaction can be viewed as an abstract coordination mechanism that allows a set of processes, each of which must be ready to participate in the interaction so that it can occur, to execute data exchange actions jointly and coordinately. (When this happens, the interaction is said to be ready for execution or *enabled*.) An attempt to participate in an interaction delays a process

until all other participants are available, and after an interaction is executed, the participants exchange some data and continue their local computations separately. Notice that an interaction being enabled does not entail its execution since it may be *linked* to other interactions in which a common process is willing to participate. Since a process can execute only one interaction at a time, an election under the linked interactions needs to be held. Intuitively, the selection procedure must be fair to avoid executions in which an interaction is never executed or never has a chance to become enabled.

A classical problem to illustrate the adequacy of multiparty interactions is the Dining Philosophers Problem. The obvious message-passing solution consists of sending requests to the forks to get them in sequence, but a deadlock may occur if each philosopher grabs the fork on his/her right, and then waits for the fork on his/her left to be released. In [16], it was proven that assuming no means of communication amongst philosophers other than through information attached to their forks, any solution in which all philosophers are programmed identically must have a possibility of deadlock. Thus, correct solutions must rely on some distinction to be made amongst the philosophers. These solutions are usually not scalable or reusable since the distinction a philosopher has to implement depends heavily on the topology of the problem. If we used multiparty interactions, the solution would be simpler since each philosopher would pick up his/her two forks at a time so that no deadlock could arise.

Figure 1 shows an $\mathcal{MI}$-based dining philosophers system based on the IP language [1]. (A brief introduction to IP is presented in Appendix A.) The philosophers are represented by processes P_i, and the forks by processes F_i ($i \in [1 \ldots N]$). Each P_i first tries to get its forks by participating in the three-party interaction Get_i together with F_i and F_{i-1}. (We assume that subindex arithmetic is module N.) Thus, acquiring a resource is specified as synchronizing with the corresponding processes in a multiparty interaction. After P_i has got its forks, it eats, releases the forks, spends some time thinking and the whole process is repeated once again. Notice that interactions Get_{i-1}, Get_i and Get_{i+1} are linked for every $i \in [1 \ldots N]$, but only one of them can be executed at the same time. The only way to guarantee that each interaction that is enabled sufficiently often shall eventually be selected for execution consists of assuming that the underlying selection mechanism is fair. (It is known that fairness is mandatory in systems in which processes need mutual exclusion to a resource [17].)

2.2. Fairness

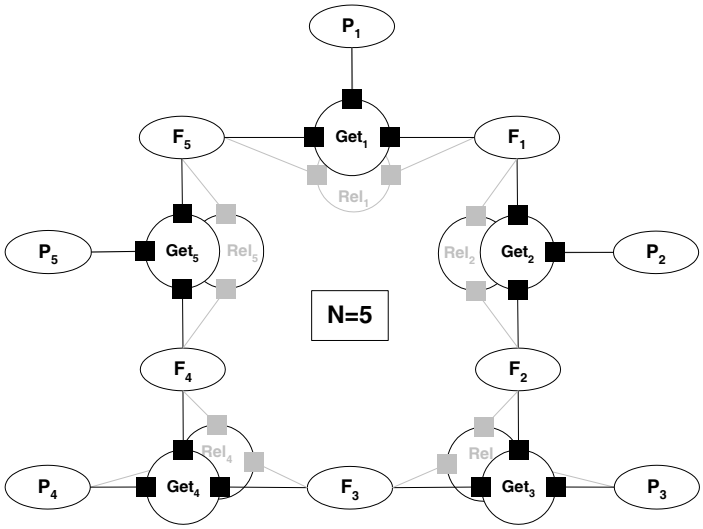
In [18], the authors introduced several properties that deserve special attention in the context of concurrent systems. They classified them into two groups, namely: safety properties, which assert that 'something bad' does not happen, and liveness properties, which assert that 'something good' must happen eventually. In concurrent programming, usual bad things are deadlocks or the violation of critical regions. In contrast, usual good things are the absence of starvation, the response to a request for service or termination.

Fairness is sometimes the only way to guarantee these liveness properties, but, unfortunately, there is not a single prevailing definition. Many authors [1–3], however, agree in that strong fairness ($\mathcal{SF}$) deserves attention since it may induce desirable liveness properties. Technically, an execution is strongly fair iff every interaction that is enabled infinitely often is executed infinitely often. This prevents an interaction that is enabled from time to time, not necessarily permanently, from being neglected. For instance, in the system in Figure 1, the only way to guarantee that each philosopher is able to eat as much as the rest is by assuming that the underlying scheduler is fair, i.e. this is the only way to insure that every request for service a philosopher makes to a fork is satisfied eventually.

DINNER :: $[||_{i=1}^{N} \mathsf{P}_i \,||\, \mathsf{F}_i]$, **where**

P_i :: *[$\mathsf{Get}_i\,\langle\rangle \rightarrow$ *eat*; $\mathsf{Rel}_i\,\langle\rangle$; *think*]
F_i :: *[$\mathsf{Get}_i\,\langle\rangle \rightarrow \mathsf{Rel}_i\,\langle\rangle \;\square\; \mathsf{Get}_{i+1}\,\langle\rangle \rightarrow \mathsf{Rel}_{i+1}\,\langle\rangle$].

(a)



(b)

Figure 1. A solution to the dining philosophers problem in IP: (a) the IP code to implement the system; (b) sketch of a system with five philosophers.

In spite of its adequacy in the context of $\mathcal{MI}$-based systems, strong fairness suffers from two practical problems that may lead to undesirable executions.

Fair finiteness. The first problem lies in the fact that strong fairness is a *void property* [19]. That is, the strong fairness fulfillment of an execution cannot be checked by performing finite experiments. Thus, there is no way to show that a scheduler produces strongly fair executions by analysing the results of an experiment. Furthermore, every finite execution is strongly fair by default [2].

Figure 2(a) shows an event trace of the system in Figure 1 ($N = 5$). Notice that interaction Get_2 is enabled n times, but it is never selected during this execution. Given that strong fairness is a void property, there is no finite experiment from which we can conclude that the scheduler that produced this execution is not strongly fair. This implies that strong fairness may lead to a situation in which an interaction is never selected in a long-enough finite execution [4].

$$P_1.\{Get_1\},\ P_2.\{Get_2\},\ F_2.\{Get_2,\ Get_3\},$$
$$(\ F_5.\{Get_5,\ Get_1\},\ F_1.\{Get_1,Get_2\},\ \mathbf{Get}_1,$$
$$P_1.\{Rel_1\},\ F_1.\{Rel_1\},\ F_5.\{Rel_1\},\ \mathbf{Rel}_1,\ P_1.\{Get_1\}\ )^n$$

(a)

$$P_1.\{Get_1\},\ P_2.\ \{Get_2\},\ P_3.\{Get_3\},$$
$$(\ F_5.\{Get_5,\ Get_1\},\ F_3.\{Get_3,\ Get_4\},\ F_1.\{Get_1,\ Get_2\},\ \mathbf{Get}_1,$$
$$F_2.\{Get_2,\ Get_3\},\ \mathbf{Get}_3,\ P_1.\{Rel_1\},\ F_1.\{Rel_1\},\ F_5.\{Rel_1\},\ \mathbf{Rel}_1,$$
$$P_3.\{Rel_3\},\ F_2.\{Rel_3\},\ F_3.\{Rel_3\},\ \mathbf{Rel}_3,\ P_1.\{Get_1\},\ P_3.\{Get_3\}\ )^\infty$$

(b)

Figure 2. Problems with strong fairness: (a) fair finiteness; (b) conspiracies. ($p.\chi$ means that process p offers to participate in any interaction in set χ, and x that interaction x is executed.)

S :: [P || Q], **where**

P :: *[A⟨⟩ → B⟨⟩ □ C⟨⟩ → skip]
Q :: *[A⟨⟩ → [B⟨⟩ → skip □ C⟨⟩ → skip]].

Figure 3. A program with inherent conspiracies.

Conspiracies. Furthermore, a scheduler may lead to executions in which all of the processes that may participate in an interaction are ready to participate in it from time to time, but it never becomes enabled because of an unfortunate interleaving that prevents them from offering to participate at the same time, i.e. some participating processes decide to execute another interaction before the former becomes enabled. These situations are commonly referred to as *conspiracies* [20,21].

Figure 2(b) shows a good example in which interaction Get_2 is readied by all of its participants infinitely many times, but never gets enabled. Although the execution is strongly fair, this conspiracy is undesirable and should be avoided. There are programs, however, in which conspiracies are inherent. For instance, the event traces of the program in Figure 3 are of the form $(P.\{A, C\}, Q.\{A\}, A, P.\{B\}, Q.\{B, C\}, B)^\infty$. The conspiracy against C is unavoidable since it is inherent to this program.

Individually, these problems have been studied by several authors [4–7, 22], giving rise to new fairness notions. Amongst them, we focus on *finitary (strong) fairness* [4] and *(strong) hyperfairness* [5] because these approaches focus on concurrent programming, whereas the others focus on self-stabilizing algorithms [6], classical temporal logic [7] and temporal logic of actions [3,22].

Table I. Comparison with related work.

Notion	Brief description	Fair finiteness	Conspiracies
$\mathcal{SF}$	Every interaction that becomes enabled infinitely often is selected infinitely often.	No	No
$\mathcal{HF}$	Every interaction that is offered infinitely often by all of its participants becomes enabled infinitely often.	No	Yes
$\mathcal{FF}$	Every interaction that becomes enabled infinitely often is selected at least once every k times it is enabled.	Yes	No
$\mathcal{CFF}_k$	No interaction is selected more than k times without analysing the state of the interactions that are linked to it.	Yes	Yes

Finitary fairness ($\mathcal{FF}$). Alur *et al.* [4] solved the finiteness problem and provided us with a new notion that needs to be combined with others. If it is combined with strong fairness, then the term 'infinitely often' is replaced by 'at least once every k times', where k is a natural number that must exist, but is not known *a priori*. Therefore, it is said that an execution is finitarily strong fair iff there exists a natural number k such that no interaction is rejected more than k times consecutively.

This notion has several drawbacks, namely (i) k is known *a posteriori*, which implies that it cannot be set *a priori* to regulate a system; (ii) its implementation is transformational; (iii) it does not attempt to solve conspiracies or alleviate them since the authors do not focus on $\mathcal{MI}$-based systems.

Hyperfairness ($\mathcal{HF}$). Attie *et al.* [5] studied conspiracies in the context of the IP language and defined hyperfairness to solve it. It is said that an execution is hyperfair iff every interaction is conspiracy-resistant, i.e. it is offered by all of their participants infinitely often. Notice that this notion insures that an interaction that can eventually become enabled, becomes enabled, which does not necessarily entail it is selected for execution; thus, it needs to be combined with other notions.

It has some drawbacks, namely (i) the set of interactions that are conspiracy-resistant needs to be pre-computed, but the authors do not provide us with an algorithm to do so; (ii) its implementation is transformational, but the authors do not provide us with a generic algorithm to perform transformations; (iii) the authors combine it with strong fairness only, which does not solve the fair finiteness problem.

2.3. Our contributions

The main contribution we present in this article consists of a new fairness notion that is more restrictive than strong fairness and addresses both the fair finiteness problem and conspiracies simultaneously, as we show in Table I. We refer to this notion as *k-conspiracy-free fairness* or $\mathcal{CFF}_k$ for short.

We think that previous attempts to solve these problems have not addressed them simultaneously since they focused on different settings. For instance, the work by Alur *et al.* on finitary fairness focuses on concurrent systems that are not $\mathcal{MI}$-based; thus, no conspiracy situations may occur. The work by Attie *et al.* focuses on $\mathcal{MI}$-based systems and it laid the foundations of hyperfairness; unfortunately, their ideas were not developed to their full extent but they deserve attention since they were the first to identify the problem and devise a solution. Very recently, Lamport [3,22] considers hyperfairness a corner-stone of action-based concurrent systems and recognizes the need for further research on this topic. Hyperfairness does not focus on fair finiteness since the authors were not interested in solving a practical problem, but in providing a notion to preserve a property called equivalence robustness, which is mandatory for a notion to be fully adequate according to the criteria in [2,23]. Conspiracies constitute a major obstacle to preserving this property, so they should be avoided.

Our proposal builds on previous theoretical work by these authors and addresses both problems from a practical standpoint since we are not interested in proving theoretical properties, but on materializing those concepts into a notion that solves practical problems. As we prove in Section 6, the implementation of our notion performs comparably to other researchers' implementations of strong fairness; however, since our notion depends on the value we assign to k beforehand, this parameter may help us control how well it addresses both problems. The parameter can thus be seen as a trade-off between effectiveness and efficiency: the smaller the value of k, the better the control of the conspiracies, but the less efficient the implementation; the greater the value of k, the poorer the control of the conspiracies, but the more efficient the implementation.

Furthermore, the implementation of our proposal is not transformational, which may be seen as a practical advantage since it can be applied to any $\mathcal{MI}$-based system without requiring us to change its source code to transform it into an equivalent fair system. Roughly speaking, we can produce a generic scheduler that can be used in any $\mathcal{MI}$-based system, whereas other proposals need to be adapted to particular systems and transform their source code to produce *ad hoc* schedulers. From a theoretical standpoint, both approaches are sound, but from a practical standpoint having a generic scheduler seems to be a better idea; otherwise, we would need to have access to the source code to transform it, which is impossible if we are dealing with processes obtained from a component which is available in binary form only, e.g. a C++ library or a CORBA object.

3. A THEORETICAL FRAMEWORK TO DESCRIBE $\mathcal{MI}$-BASED SYSTEMS

In this section, we present the foundations we need to define our notion rigorously, which we think is very important so that other authors can repeat our work. Later, we show that the framework we have designed allows us to describe other authors' notions. Thus, the implementation we present in Section 5 can be seen as a generic harness to implement $\mathcal{MI}$-based systems and fairness notions.

3.1. Definitions

The core of the framework is a set of definitions with which we define rigorously the concepts presented previously. We use the dining philosophers system in Figure 1 to illustrate some of them.

Definition 1. ($\mathcal{MI}$ systems) A system Σ is a 2-tuple of the form (P_Σ, I_Σ) in which $P_\Sigma \neq \emptyset$ is a finite set of processes and $I_\Sigma \neq \emptyset$ is a finite set of interactions. We denote the set of processes that may eventually offer to participate in interaction x as $\mathbb{P}(x)$, and the set of interactions that process p can offer as $\mathbb{I}(p)$.

In our example, we have an $\mathcal{MI}$-based system composed of $N = 5$ philosopher processes called $\mathsf{P_i}$ and N fork processes called $\mathsf{F_i}$ ($i \in [1 \ldots N]$). These processes are synchronized by means of N interactions called $\mathsf{Get_i}$ to take the forks and five interactions called $\mathsf{Rel_i}$ to release them. For instance, the set of processes participating in interaction $\mathsf{Get_i}$ is $\mathbb{P}(\mathsf{Get_i}) = \{\mathsf{F_{i-1}}, \mathsf{P_i}, \mathsf{F_i}\}$, and the set of interactions in which $\mathsf{F_i}$ may participate is $\mathbb{I}(\mathsf{F_i}) = \{\mathsf{Get_i}, \mathsf{Rel_i}, \mathsf{Get_{i+1}}, \mathsf{Rel_{i+1}}\}$.

Definition 2. (Events) An event is a happening that induces a system to transit from a configuration to another. (A configuration is an object that may be viewed as a snapshot of a system at run time.) In our model, we take the following kinds of events into account.

- *Offering event.* $p.\chi$ indicates that process p is offering to participate in an interaction in set χ. Notice that if $\chi = \emptyset$, process p arrives at a fixed point that we may interpret as its termination because it can neither perform local computation nor execute any interaction.
- *Synchronization.* x indicates that interaction x has been selected for execution.

For instance, when philosopher $\mathsf{P_i}$ offers to participate in interaction $\mathsf{Get_i}$, an event of the form $\mathsf{P_i}.\{\mathsf{Get_i}\}$ occurs; similarly, when $\mathsf{P_i}$ takes its forks, an event of the form $\mathsf{Get_i}$ occurs and synchronizes the execution of $\mathsf{P_i}$, $\mathsf{F_i}$ and $\mathsf{F_{i-1}}$ ($i \in [1 \ldots N]$).

Definition 3. (Executions) An execution of system Σ is a 3-tuple (C_0, α, β) in which C_0 is the initial configuration, $\alpha = [C_1, C_2, C_3, \ldots]$ is a maximal (finite or infinite) sequence of configurations, and $\beta = [e_1, e_2, e_3, \ldots]$ is a maximal (finite or infinite) sequence of events responsible for the transition between every two consecutive configurations. (Obviously, $|\alpha| = |\beta|$.) Finally, let $\lambda = (C_0, \alpha, \beta)$ be an execution of system Σ. We call α its configuration trace and denote it as λ_α, and β its event trace and denote it as λ_β.

Consider, for instance, the execution below:

$$\lambda = (C_0, \alpha, \beta)$$
$$\alpha = [C_1, C_2, C_3, C_4, \ldots]$$
$$\beta = [\mathsf{P_1}.\{\mathsf{Get_1}\}, \mathsf{F_5}.\{\mathsf{Get_5}, \mathsf{Get_1}\}, \mathsf{F_1}.\{\mathsf{Get_1}, \mathsf{Get_2}\}, \mathsf{Get_1}, \ldots]$$

Philosopher $\mathsf{P_1}$ starts offering interaction $\mathsf{Get_1}$, fork $\mathsf{F_5}$ then offers interactions $\{\mathsf{Get_5}, \mathsf{Get_1}\}$, and fork $\mathsf{F_1}$ interactions $\{\mathsf{Get_1}, \mathsf{Get_2}\}$. Interaction $\mathsf{Get_1}$ becomes enabled at configuration C_3 and, in this case, it is executed and the program continues.

Definition 4. (Semantics) We denote the rule that captures the underlying semantics that control the transition between configurations as $\longrightarrow_{\mathrm{L}}$. For instance, $C \xrightarrow{e}_{\mathrm{L}} C'$ indicates that the system may transit from configuration C to configuration C' on occurrence of event e. Thus, given an execution $\lambda = (C_0, [C_1, C_2, C_3, \ldots], [e_1, e_2, e_3, \ldots])$, we usually write it as $C_0 \xrightarrow{\mathsf{e_1}}_{\mathrm{L}} C_1 \xrightarrow{\mathsf{e_2}}_{\mathrm{L}} C_2 \xrightarrow{\mathsf{e_3}}_{\mathrm{L}} \cdots$.

Our example is implemented in IP, thus $\longrightarrow_{\mathrm{L}}$ amounts for $\longrightarrow_{\mathrm{IP}}$. Please, consult [1] for a complete description of the semantics of the IP language or Appendix A for a brief introduction.

Definition 5. (Processes) Process p is waiting for an interaction in set $\Upsilon \neq \emptyset$ at the ith configuration in execution λ iff it has arrived at a point in which it may execute any $x \in \Upsilon$, i.e. it has offered to participate in a subset of interactions $\chi \supseteq \Upsilon$ and no interaction in Υ has been selected since that moment. Process p is finished at the ith configuration in execution λ iff it has offered to participate in an empty set of interactions, that is, it can neither perform local computations nor interact with other processes.

$$\mathsf{Waiting}(\lambda, p, \Upsilon, i) \iff \exists \chi \supseteq \Upsilon, k \in [1 \ldots i) \cdot (\lambda_\beta(k) = p.\chi \wedge \nexists j \in (k \ldots i] \cdot \lambda_\beta(j) = x \wedge x \in \Upsilon)$$
$$\mathsf{Finished}(\lambda, p, i) \iff \exists k \in [1 \ldots i] \cdot \lambda_\beta(k) = p.\emptyset$$

In our example, philosopher $\mathsf{P_j}$ is readying the set of interactions $\{\mathsf{Get_j}\}$ ($j \in [1 \ldots N]$) at configuration C_i ($i \in [1 \ldots |\lambda|]$) if an event $\mathsf{P_j}.\{\mathsf{Get_j}\}$ happened before C_i and interaction $\mathsf{Get_j}$ was not selected since that moment.

Definition 6. (Interactions) Interaction x is enabled at the ith configuration in execution λ iff all of the processes in $\mathbb{P}(x)$ are offering x at that configuration, that is, all of its participants are waiting for it to be selected. Interaction x is stable at the ith configuration in execution λ iff its participants are finished or waiting for an interaction, whichever it is.

$$\mathsf{Enabled}(\lambda, x, i) \iff \forall p \in \mathbb{P}(x) \cdot \mathsf{Waiting}(\lambda, p, \{x\}, i)$$
$$\mathsf{Stable}(\lambda, x, i) \iff \forall p \in \mathbb{P}(x) \cdot (\mathsf{Finished}(\lambda, p, i) \vee \exists \Upsilon \subseteq I_\Sigma \cdot \mathsf{Waiting}(\lambda, p, \Upsilon, i))$$

Interaction $\mathsf{Get_j}$ is enabled at the ith configuration iff all of its participants ($\mathsf{P_j}$, $\mathsf{F_j}$ and $\mathsf{F_{j-1}}$) are offering it. Furthermore, $\mathsf{Rel_j}$ is stable because its participants are waiting for $\mathsf{Get_j}$. Notice that enablement implies stableness, but the converse is not true in general.

Definition 7. (Miscellaneous) Let λ be an execution and x an interaction. We define the following sets at the ith configuration:

1. Interactions linked to x: the set of interactions that share a participant with x.
$$\mathsf{Linked}(\lambda, x, i) = \{y \in I_\Sigma \mid \mathbb{P}(x) \cap \mathbb{P}(y) \neq \emptyset\}$$
2. Set of enablements: the set of indices up to i that identify the configurations at which interaction x is enabled.
$$\mathsf{EnaSet}(\lambda, x, i) = \{k \in [0 \ldots i] \mid \mathsf{Enabled}(\lambda, x, k)\}$$
3. Set of executions: the set of indices up to i that identify the configurations at which interaction x is selected.
$$\mathsf{ExeSet}(\lambda, x, i) = \{k \in [0 \ldots i] \mid \lambda_\beta(k) = x\}$$
4. Set of offerings: the set of indices up to i that identify the configurations at which p offers interaction x.
$$\mathsf{OffSet}(\lambda, x, p, i) = \{k \in [0 \ldots i] \mid \lambda_\beta(k) = p.\chi \wedge x \in \chi\}$$

5. We also define predicates $\mathsf{InfEna}(\lambda, x)$, $\mathsf{InfExe}(\lambda, x)$, and $\mathsf{InfOff}(\lambda, x, p)$ to determine if an interaction is enabled, selected, or offered infinitely many times along an execution.

$$\begin{aligned}\mathsf{InfEna}(\lambda, x) &\iff |\,\mathsf{EnaSet}(\lambda, x, |\lambda|)| = \infty\\ \mathsf{InfExe}(\lambda, x) &\iff |\,\mathsf{ExeSet}(\lambda, x, |\lambda|)| = \infty\\ \mathsf{InfOff}(\lambda, x, p) &\iff |\,\mathsf{OffSet}(\lambda, x, p, |\lambda|)| = \infty\end{aligned}$$

3.2. Realizing the benefits

In this section, we show how to use our framework to define common fairness notions, which proves it is very flexible.

Example 1. (Strong fairness) An execution is strongly fair iff every interaction that is enabled infinitely often is selected infinitely often:

$$\mathcal{SF}(\lambda) \iff \forall x \in I_\Sigma \cdot (\mathsf{InfEna}(\lambda, x) \Rightarrow \mathsf{InfExe}(\lambda, x))$$

Example 2. (Strong hyperfairness) An execution is strongly hyperfair iff every interaction is conspiracy-resistant, i.e. it is eventually offered by all of its participants:

$$\begin{aligned}\mathcal{HF}(\lambda) &\iff \forall \in I_\Sigma \cdot \mathsf{ConspResistant}(\lambda, x)\\ \mathsf{ConspResistant}(\lambda, x) &\iff \forall p \in \mathbb{P}(x) \cdot \mathsf{InfOff}(\lambda, x, p)\end{aligned}$$

Example 3. (Finitary strong fairness) An execution is finitarily-strong fair iff there is a natural number k such that no interaction is rejected more than k consecutive times:

$$\mathcal{FF}(\lambda) \iff \exists k > 0 \cdot \forall x \in I_\Sigma, i \in \mathsf{ExeSet}(\lambda, x, |\lambda|) \cdot \sum_{j=\mathsf{PrevExe}(\lambda,x,i)}^{i} \mathsf{Rejected}(\lambda, x, j) < k$$

where $\mathsf{Rejected}$ and $\mathsf{PrevExe}$ are defined as follows (notice that we denote the maximum or minimum of an empty set as $\perp$):

$$\begin{aligned}\mathsf{Rejected}(\lambda, x, i) \iff{}& \exists j \in [0 \ldots i) \cdot (\mathsf{Enabled}(\lambda, x, j) \wedge \nexists k \in [j \ldots i] \cdot k \in \mathsf{ExeSet}(\lambda, x, j)) \wedge\\ & \neg\, \mathsf{Enabled}(\lambda, x, i)\end{aligned}$$

$$\mathsf{PrevExe}(\lambda, x, i) = \begin{cases} j & \text{if } j = \max \mathsf{ExeSet}(\lambda, x, i-1) \wedge j \neq \perp \\ 1 & \text{otherwise} \end{cases}$$

4. k-CONSPIRACY-FREE FAIRNESS

According to Attie *et al.*'s definition [5], a conspiracy against interaction x happens iff every participant of x offers it infinitely many times, but it never gets enabled because some of them commit to another interaction too rashly. Thus, if we need to avoid these situations, we have to insure that the interactions that are linked with x are not selected if any of them is not stable. Formally, we can define a potential conspiracy situation as follows.

Definition 8. (Potential conspiracy) A potential conspiracy against x occurs at the ith configuration of execution λ iff predicate $\mathsf{PotentialConsp}(\lambda, x, i)$ holds:

$$\mathsf{PotentialConsp}(\lambda, x, i) \iff \exists y \in \mathsf{Linked}(\lambda, x, i) \cdot \neg\, \mathsf{Stable}(\lambda, y, i)$$

With this predicate, we can define a new notion to insure the absence of conspiracies:

$$\forall x \in I_\Sigma, i \in \mathsf{ExeSet}(\lambda, x, i) \cdot \neg\, \mathsf{PotentialConsp}(\lambda, x, i)$$

However, this has an important drawback since it forces all linked interactions to be stable before selecting one of them, but it does not insure that an interaction that is enabled infinitely many times is selected infinitely many times. Thus, we can strengthen this formula as follows:

$$\mathcal{SF}(\lambda) \wedge \forall x \in I_\Sigma, i \in \mathsf{ExeSet}(\lambda, x, i) \cdot \neg\, \mathsf{PotentialConsp}(\lambda, x, i)$$

Unfortunately, the executions to which this formula can be applied must be infinite, which does not solve the finiteness problem. A proposal for implementing strong notions in finite executions using queues was presented in [24,25]. The authors demonstrated that arranging interactions using priority queues results in strongly fair executions. Thus, if we arrange interactions in a queue according to their age (the time that has elapsed since they were selected for the last time or ∞ if they have never been selected) and select the oldest one, we can insure strong fairness:

$$\forall x \in I_\Sigma, i \in \mathsf{ExeSet}(\lambda, x, i) \cdot (\neg\, \mathsf{PotentialConsp}(\lambda, x, i) \wedge \mathsf{Oldest}(\lambda, x, i))$$

where Oldest is defined as follows:

$$\mathsf{Oldest}(\lambda, x, i) \iff \forall y \in \mathsf{Linked}(\lambda, x, i) \cdot \mathsf{Age}(\lambda, x, i) \geq \mathsf{Age}(\lambda, y, i)$$

$$\mathsf{Age}(\lambda, x, i) = \begin{cases} i - \mathsf{PrevExe}(\lambda, x, i) & \text{if } \mathsf{PrevExe}(\lambda, x, i) \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

Since we do not take real-time constraints into account, the age of an interaction can be modelled as the number of configurations a program has reached since the last time an interaction was selected or ∞ if it has never been selected before.

The previous formula is too restrictive because it forces linked interactions to wait for each other continuously. Thus, the pace at which a group of linked interactions progresses is determined by the slowest process that participates in any of them. To improve this, we introduce the concept of *conspiracy threshold* and define it as the maximum number of times we allow an interaction to be selected without waiting for the interactions linked to it to become stable. An execution is thus said to be *k-conspiracy-free fair* iff an interaction is executed as long as (i) it is not in a potential conspiracy situation and it is the oldest or (ii) it is in a potential conspiracy situation, but it has not been selected more than k times between two consecutive non-potential conspiracy situations.

Definition 9. (k-conspiracy-free fairness) Execution λ is k-conspiracy-free fair for any $k \geq 0$ iff predicate $\mathcal{CFF}_k(\lambda)$ holds:

$$\begin{aligned} \mathcal{CFF}_k(\lambda) \iff \forall x \in I_\Sigma, i \in \mathsf{ExeSet}(\lambda, x, |\lambda|) \cdot \\ (\neg\, \mathsf{PotentialConsp}(\lambda, x, i) \wedge \mathsf{Oldest}(\lambda, x, i)) \vee \\ \sum_{j=\mathsf{PrevNotConsp}(\lambda,x,i)}^{i} \mathsf{PotentialConsp}(\lambda, x, j) < k \end{aligned}$$

where PrevNotConsp is defined as follows

$$\mathsf{PrevNotConsp}(\lambda, x, i) = \begin{cases} j & \text{if } j = \max\{k \in (0 \ldots i) \mid \mathsf{PotentialConsp}(\lambda, x, k)\} \wedge j \neq \bot \\ 1 & \text{otherwise} \end{cases}$$

Next, we sketch the arguments that allow us to claim that this notion addresses fair finiteness and conspiracies. They are also supported by the experimental results in Section 6.

Fair finiteness. k may be viewed as a conspiracy threshold, i.e. the maximum number of times the non-determinism embodied in a program may conspire against an interaction. For instance, the execution in Figure 2(a) is not k-conspiracy-free fair for any $n > k$ because interaction Get_1 executes n times without waiting for Get_5 to be stable. Thus, not every finite execution satisfies this criterion.

Conspiracies. In the execution in Figure 2(b) our notion would not allow Get_1 or Get_3 to be selected once they have been executed k times unless the interactions linked to it are known to be stable, thus giving them a chance to become enabled. How often this happens depends on k, which thus can be used to control the goodness of this notion to solve conspiracies.

Parameter k is known and needs to be set *a priori* to fine-tune the notion according to the characteristics of the system under consideration. Thus, if $k = 0$ (its minimum value) the pace at which a group of linked interactions progresses is determined by the slowest process that participates in any of them, i.e. it implements a round-robin strategy within groups of linked interactions. In contrast, if $k \to \infty$, interactions are selected as soon as possible once they are detected to be enabled, i.e. it implements a best-effort strategy.

It is also important to notice that $\mathcal{CFF}_k$ is not a void property [19] since not every finite execution satisfies the criterion as was the case for strong fairness or strong hyperfairness. Therefore, we can prove experimentally whether a scheduler implements it or not by means of finite experiments, which is very important from a practical standpoint.

5. IMPLEMENTING OUR FRAMEWORK AND $\mathcal{CFF}_k$

Our framework is quite expressive, thus a trivial implementation in which the whole trace of a program is maintained in memory would not be efficient enough. In this section we report on how to implement it by means of several abstract data types and rules that allow the implementation to record just the information it needs to implement the framework at the current configuration. As we prove in Section 6, this is both efficient and effective.

5.1. On the implementation of the framework

Figure 4(a) shows the data structures we need to implement our framework.

- A readiness map (φ): maps the set of interaction I_Σ onto subsets of processes. $\varphi(x)$ denotes the set of processes that are offering interaction x at the current configuration. Initially, φ maps every interaction to $\emptyset$.

$$\varphi : I_\Sigma \longrightarrow 2^{P_\Sigma} \ (\mathrm{dom}\,\varphi = I_\Sigma)$$
$$\vartheta \subseteq P_\Sigma$$
$$\gamma, \delta, \epsilon : I_\Sigma \longrightarrow \mathbb{N} \ (\mathrm{dom}\,\gamma = \mathrm{dom}\,\delta = \mathrm{dom}\,\epsilon = I_\Sigma)$$

(a)

$$\texttt{AddOffer}(\varphi, p, \chi) = \{x \mapsto \varphi(x) \mid x \notin \chi\} \cup \{x \mapsto \varphi(x) \cup \{p\} \mid x \in \chi\}$$
$$\texttt{RemoveOffer}(\varphi, x) = \{y \mapsto \varphi(y) \setminus \mathbb{P}(x) \mid y \in I_\Sigma\}$$
$$\texttt{AddFinished}(\vartheta, p) = \vartheta \cup \{p\}$$
$$\texttt{IncOff}(\gamma, \chi) = \{x \mapsto \gamma(x) \mid x \notin \chi\} \cup \{x \mapsto \gamma(x) + 1 \mid x \in \chi\}$$
$$\texttt{IncEna}(\delta, \varphi) = \{x \mapsto \delta(x) \mid \varphi(x) \neq \mathbb{P}(x)\} \cup \{x \mapsto \delta(x) + 1 \mid \varphi(x) = \mathbb{P}(x)\}$$
$$\texttt{IncExe}(\epsilon, x) = \{y \mapsto \epsilon(y) \mid y \in I_\Sigma \setminus \{x\}\} \cup \{x \mapsto \epsilon(x) + 1\}$$

(b)

Figure 4. Abstract data types to implement our framework: (a) data structures; (b) updating functions.

- A set of processes that are finished (ϑ): it is a set of processes that can neither execute local computations nor offer any interaction, i.e. they are finished. It is initialized to $\emptyset$.
- A map of offers (γ): maps I_Σ onto natural numbers that count the number of times an interaction has been offered. Initially, γ maps every interaction to 0.
- An enablement map (δ): maps I_Σ onto natural numbers that count the number of times interaction x has been enabled. Initially, δ maps every interaction to 0.
- An execution map (ϵ): maps I_Σ onto natural numbers that count the number of times each interaction has been executed. Initially, ϵ maps every interaction to 0.

Figure 4(b) shows a rigorous definition of the functions we need to manipulate previous data structures.

- Function $\texttt{AddOffer}(\varphi, p, \chi)$ is used to update φ every time an event of the form $p.\chi$ occurs ($\chi \neq \emptyset$). It returns a map in which process p is added to the readiness map of every interaction x in χ.
- Function $\texttt{RemoveOffer}(\varphi, x)$ is used to remove the participants of interaction x from φ every time it is executed.
- Function $\texttt{AddFinished}(\vartheta, p)$ is used to update ϑ when process p finishes.
- Functions $\texttt{IncOff}(\gamma, \chi)$ and $\texttt{IncEna}(\delta, \varphi)$ are used to increase the counters of offers and enablements, respectively, when an event of the form $p.\chi$ occurs ($\chi \neq \emptyset$). Analogously, $\texttt{IncExe}(\epsilon, x)$ updates map ϵ every time an event of the form x occurs.

In Table II, we describe how the functions and predicates of the framework can be implemented by means of the data structures and functions mentioned above. For the sake of readability, every function or predicate from our framework referred to as X (using the sans serif font) has a counterpart in the implementation referred to as `X` (using the `typewriter` font).

Table II. Mapping framework functions and predicates onto their implementation.

Theoretical framework	Implementation at the current configuration
Waiting$(\lambda, p, \Upsilon, i)$	$\texttt{Waiting}(\varphi, \Upsilon, p) \iff \forall x \in \Upsilon \cdot p \in \varphi(x)$
Finished(λ, p, i)	$\texttt{Finished}(\vartheta, p) \iff p \in \vartheta$
Enabled(λ, x, i)	$\texttt{Enabled}(\varphi, x) \iff \varphi(x) = \mathbb{P}(x)$
Stable(λ, x, i)	$\texttt{Stable}(\varphi, x) \iff \forall p \in \mathbb{P}(x) \cdot (p \in \bigcup \text{img}\, \varphi \vee p \in \vartheta)$
Linked(λ, x, i)	$\texttt{Linked}(\varphi, x) = \{y \in I_\Sigma \mid \mathbb{P}(x) \cap \mathbb{P}(y) \neq \emptyset\}$

$$\frac{C \xrightarrow{p.\emptyset}_{\text{L}} C' \wedge \vartheta' = \texttt{AddFinished}(\vartheta, p)}{(C, \varphi, \vartheta, \gamma, \delta, \epsilon) \xrightarrow{p.\emptyset}_{\texttt{FW}} (C', \varphi, \vartheta', \gamma, \delta, \epsilon)} \tag{1}$$

$$\frac{\begin{array}{c} C \xrightarrow{p.\chi}_{\text{L}} C' \wedge \chi \neq \emptyset \wedge \\ \varphi' = \texttt{AddOffer}(\varphi, p, \chi) \wedge \gamma' = \texttt{IncOff}(\gamma, \chi) \wedge \delta' = \texttt{IncEna}(\delta, \varphi') \end{array}}{(C, \varphi, \vartheta, \gamma, \delta, \epsilon) \xrightarrow{p.\chi}_{\texttt{FW}} (C', \varphi', \vartheta, \gamma', \delta', \epsilon)} \tag{2}$$

$$\frac{\begin{array}{c} C \xrightarrow{x}_{\text{L}} C' \wedge \\ \varphi' = \texttt{RemoveOffer}(\varphi, x) \wedge \epsilon' = \texttt{IncExe}(\epsilon, x) \end{array}}{(C, \varphi, \vartheta, \gamma, \delta, \epsilon) \xrightarrow{x}_{\texttt{FW}} (C', \varphi', \vartheta, \gamma, \delta, \epsilon')} \tag{3}$$

Figure 5. The semantics of our framework.

Once we have the adequate data structures, their corresponding updating functions, and an implementation of the functions and predicates of our framework, we can define its semantics, i.e. the rules that control how it reacts on occurrence of an event. An informal explanation of the rules in Figure 5 follows.

- Rule 1 describes how our framework reacts on occurrence of an event of the form $p.\chi$ in which $\chi = \emptyset$. In this case, we use function $\texttt{AddFinished}(\vartheta, p)$ to update set ϑ, thus recording that process p is finished.
- Rule 2 describes how the framework reacts to an event of the form $p.\chi$ in which $\chi \neq \emptyset$. On occurrence of such an event, it updates φ to record there is a new set of offers that might enable one or more interactions, γ to record how many times they have been offered, and δ to record how many times they have been enabled. Note that δ' is calculated on the next configuration since we apply $\texttt{IncEna}$ to δ and φ'. This way, we can calculate the new set of interactions that are enabled once the event that fires this rule has occurred.
- Rule 3 describes how our framework reacts if an enabled interaction is selected for execution. It is fired if the underlying language allows interaction x to be executed in the current configuration; in this case, the framework removes its participants from the image of map φ (this might disable some interactions) and increases the execution counter associated with it by one.

$$\tau : \mathbb{N} \longrightarrow I_\Sigma \ (\text{img}\,\tau = I_\Sigma)$$
$$\rho : I_\Sigma \longrightarrow \mathbb{N} \ (\text{dom}\,\rho = I_\Sigma)$$

(a)

$$\texttt{MoveRear}([x_1, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots x_n], x_j) = [x_1, \ldots, x_{j-1}, x_{j+1}, \ldots x_n, x_j]$$
$$\texttt{Oldest}(\tau, \varphi) = x \iff \nexists y \neq x \cdot (\texttt{Enabled}(\varphi, y) \wedge \tau^{-1}(y) < \tau^{-1}(x))$$
$$\texttt{IncreaseConsp}(\rho, x) = \{y \mapsto \rho(y) \mid x \neq y\} \cup \{x \mapsto \rho(x) + 1\}$$
$$\texttt{ResetConsp}(\rho, x) = \{y \mapsto \rho(y) \mid x \neq y\} \cup \{x \mapsto 0\}$$
$$\texttt{PotentialConsp}(\varphi, x) \iff \exists y \in \texttt{Linked}(\varphi, x) \cdot \neg\, \texttt{Stable}(\varphi, y)$$

(b)

Figure 6. Abstract data types to implement $\mathcal{CFF}_k$: (a) data structures; (b) supporting functions and predicates.

Note that these rules select an arbitrary interaction that is ready for execution according to the semantics of the underlying language ($\longrightarrow_{\text{L}}$), and that the non-determinism embodied in the application of these rules contributes to the non-determinism of the program being executed. However, no rule can be applied continuously to the detriment of the others: on the one hand, Rules 1 and 2 may not be applicable continuously since the number of processes that can stop or offer interaction is finite; thus, after a finite number of applications, Rule 3 must necessarily be applicable or the system deadlocks. On the other hand, Rule 3 cannot be applied continuously since a new interaction cannot become enabled if its participants do not have an opportunity to offer it. This is the reason why the implementation of these rules need not rely on any fairness assumption to be sound.

5.2. On the implementation of $\mathcal{CFF}_k$

To implement $\mathcal{CFF}_k$, we first need to define two additional data structures and their corresponding updating functions and predicates, as shown in Figure 6. Next, we present an explanation of these data structures.

- A queue of interactions (τ): we arrange the set of interactions in a queue so that the closer they are to the rear, the less time has elapsed since they were executed for the last time. We denote the position of interaction x in this queue as $\tau^{-1}(x)$. Initially, the interactions may be arranged arbitrarily since none of them has been selected previously.
- A map of potential conspiracy counters (ρ): we use this map to count the number of times any interaction linked to x has been selected in configurations in which x was not stable. Initially, ρ maps every interaction to 0.

Next, we present an explanation of the updating functions and the predicates we need.

- Function $\texttt{MoveRear}(\tau, x)$: moves x to the rear of τ, but preserves the relative position of the remaining interactions.

$$\frac{D \xrightarrow{p.\emptyset}_{\texttt{FW}} D'}{(D, \tau, \rho) \xrightarrow{p.\emptyset}_{\texttt{CFF}_\texttt{k}} (D', \tau, \rho)} \tag{4}$$

$$\frac{D \xrightarrow{p.\chi}_{\texttt{FW}} D'}{(D, \tau, \rho) \xrightarrow{p.\chi}_{\texttt{CFF}_\texttt{k}} (D', \tau, \rho)} \tag{5}$$

$$\frac{\begin{array}{c} D \xrightarrow{x}_{\texttt{FW}} D' \wedge \\ (\neg\, \texttt{PotentialConsp}(\varphi, x) \wedge x = \texttt{Oldest}(\tau, \varphi) \wedge \rho' = \texttt{ResetConsp}(\rho, x) \vee \\ (\texttt{PotentialConsp}(\varphi, x) \wedge \rho(x) < \texttt{k} \wedge \rho' = \texttt{IncreaseConsp}(\rho, x)) \wedge \\ \tau' = \texttt{MoveRear}(\tau, x) \wedge \end{array}}{(D, \tau, \rho) \xrightarrow{x}_{\texttt{CFF}_\texttt{k}} (D', \tau', \rho')} \tag{6}$$

where $D = (C, \varphi, \vartheta, \gamma, \delta, \epsilon)$.

Figure 7. An implementation of $\mathcal{CFF}_k$.

- Function $\texttt{Oldest}(\tau, \varphi)$: the implementation of $\mathsf{Oldest}(\lambda, x, i)$ when C_i is the current configuration. It returns the first enabled element in queue τ.
- Function $\texttt{IncreaseConsp}(\rho, x)$: increases the potential conspiracy counter of interaction x when it is selected and there is a non-stable, linked interaction.
- Function $\texttt{ResetConsp}(\rho, x)$: resets the potential conspiracy counter associated with x.
- Predicate $\texttt{PotentialConsp}(\varphi, x)$: a trivial implementation of $\mathsf{PotentialConsp}(\lambda, x, i)$ at the current configuration.

Once we have defined these supporting structures, functions and predicates, we can implement $\mathcal{CFF}_k$ by means of the rules shown in Figure 7. Note that these rules work on configurations of the form $E = (D, \tau, \rho)$, where $D = (C, \varphi, \vartheta, \gamma, \delta, \epsilon)$ is a configuration on which the $\longrightarrow_{\texttt{FW}}$ can work. Next, we describe them intuitively.

- Rules 4 and 5 are trivial, since every time the framework reacts to an event of the form $p.\chi$, there is nothing to do except to record the structures that the framework has updated.
- Rule 6 is also straightforward since it allows us to decide if an enabled interaction fulfills our selection criterion. Note the close correspondence between this rule and the definition in Section 4: an interaction may be selected as long as (i) it is not in a potential conspiracy situation and it is the oldest, or (ii) it is in a potential conspiracy situation but its conspiracy counter has not exceeded k (the conspiracy threshold). In the former case, the conspiracy counter associated with the interaction selected is reset, but increased in the latter. In both cases, the interaction selected is moved to the rear of queue τ.

6. PERFORMANCE

In order to evaluate our framework and our notion, we measured their performance and effectiveness using the dining philosophers system in Figure 1. We implemented it using the J# programming language, which is an efficient Java dialect for the .NET platform [26], and we ran our tests on a 2.0 GHz AMD Athlon XP machine equipped with 512 MB of DDR 266 MHz memory. We assumed that the time each philosopher spends at thinking or eating is negligible with respect to the time needed to detect enablements, get mutual exclusion or select interactions, for instance. In this setting, each philosopher should be able to have lunch as much as the others during a long-enough fair execution.

Each experiment consisted of a system composed of N philosophers and N forks ($N = 10, 20, \ldots, 100$). We terminated the experiments after executing 10 000 interactions, i.e. 5000 Get and 5000 Rel interactions were executed in each experiment. They were run 100 times, and we computed the average value of the following metrics.

1. Execution time: the average time to execute 10 000 interactions.
2. Selection time: the average time an interaction needs to be selected since it was offered for the first time by one of its participants.
3. Rejection ratio: the percentage of rejections with regard to the number of interactions executed, i.e. the number of times an enabled interaction becomes disabled because another interaction linked to it is selected for execution.

We compared our proposal with the random selection criterion the framework itself uses to select an enabled interaction, the counter-based proposal by Francez and Forman [1], and the incremental one by Corchuelo *et al.* [24]. The results using Best's [20,21] or Olderog and Apt's algorithms [27] were so similar to the results using Francez and Forman's algorithm that we decided not to show them explicitly. The proposals by Joung [28,29] are so costly in practice that the times we obtained exceeded the rest by orders of magnitude.

Figure 8 shows that the random proposal is the fastest one, whereas the slowest one is $\mathcal{CFF}_k$ when $k = 1$. Note that the incremental proposal performs better than Francez and Forman's since it needs not examine the whole set of interactions before reaching a decision, but not as well as the random proposal since the data structures it needs to maintain are more complex. The execution time of $\mathcal{CFF}_k$ depends on the value we assign to k. If k is small ($k = 1$), it performs slightly worse than Francez and Forman's proposal because it amounts to a round-robin strategy in which interactions are executed in rounds, but the algorithm is more complex. However, if k is high ($k = 10\,000$), its performance is similar to the random proposal, although it cannot keep alleviating conspiracies as well as before.

This is the behaviour we expected since k constrains the number of times that linked interactions have to wait for each other; thus, the smaller the value of k, the more they have to wait for each other. If we have 100 philosophers, the average number of interactions per second ranges from 109 int/s in the worst case ($\mathcal{CFF}_k$ with $k = 1$) to 335 int/s in the best case (the random proposal).

Figure 9 shows the time the random proposal spends at selecting interactions, which is obviously faster than that of Francez and Forman. The selection time of our algorithm depends on k. That is, in the worst case ($k = 1$) it is better than Francez and Forman's proposal, but its execution time is worse because the cost of updating our data structures is higher. Finally, if $k = 10\,000$ our proposal behaves like the random proposal because almost no interaction reaches the conspiracy threshold and those that are linked almost never have to wait for each other.

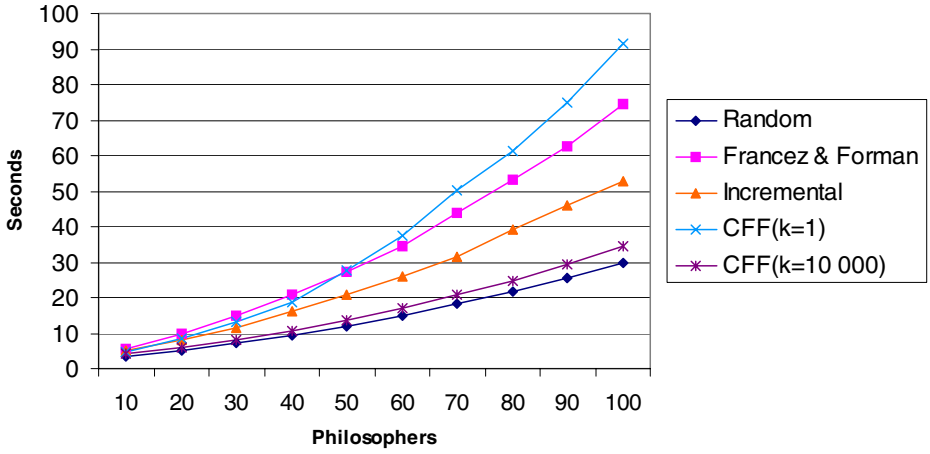


Figure 8. Execution times.

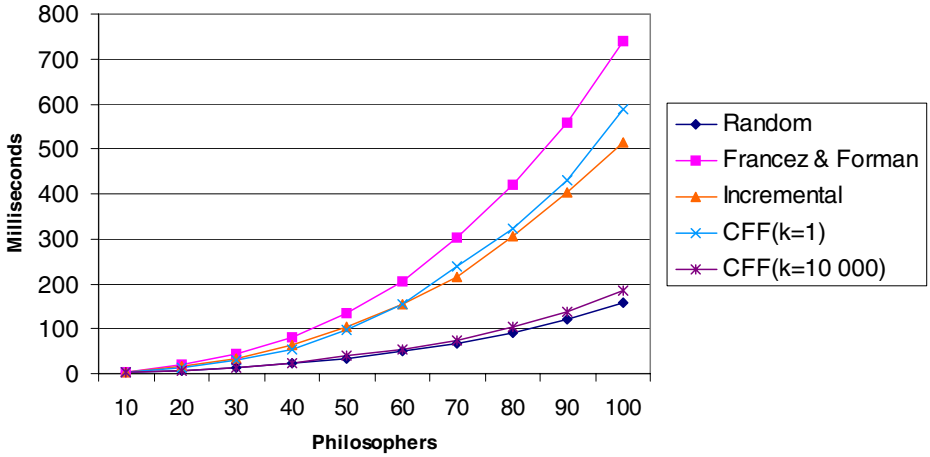


Figure 9. Selection times.

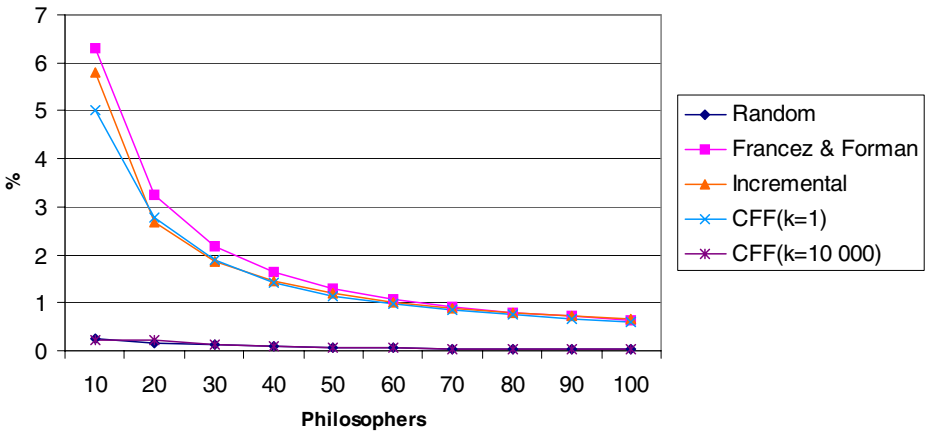


Figure 10. Rejection ratios.

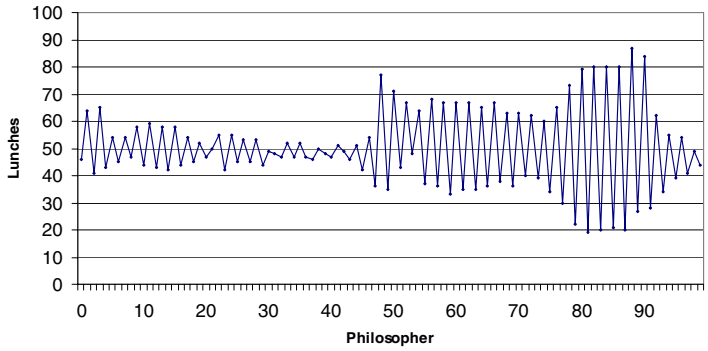
When we have 100 philosophers, the time needed to select interactions in Francez and Forman's proposal is slightly greater than 700 ms, i.e. the time that the philosophers need to offer to participate in interactions in which they are interested. Depending on the value of k, the selection time of our algorithm ranges from 186 ms in the best case ($k = 10\,000$) to 589 ms in the worst case ($k = 1$).

Figure 10 shows how the rejection ratio decreases when the number of philosophers increases. If there are 10 philosophers, the rejection ratio ranges from 0.3% to 6.3%. Note that 0.3% is the minimum rejection ratio because the philosophers need to get mutual exclusion with their neighbours to get their forks. These results corroborate the behaviour of the selection time since a rejection implies that the rejected interaction has to be offered again.

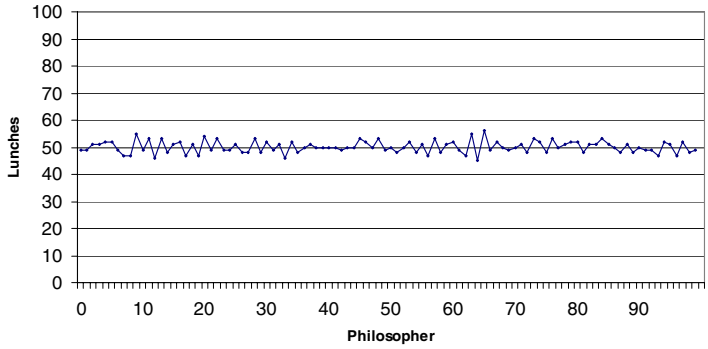
We also counted the number of times each philosopher had lunch, i.e. the number of Get interactions that were executed. The data plotted in Figure 11 allow us to detect potential conspiracy situations because philosopher i is almost permanently interested in interaction Get_i, which is linked to interactions Get_{i-1} and Get_{i+1}. Therefore, if the variations in the distribution of lunches between every three consecutive philosophers is high, this means that the execution is not equitable. In a scenario such as ours, in which the philosophers eat and think for the same, negligible amount of time, equity is obviously desirable.

Figure 11(a) shows the distribution of lunches using the random proposal. There is a large difference between the number of times each philosopher eats. For instance, philosopher P_{86} had 80 lunches, whereas his/her neighbours had 20 (P_{85}) and 21 (P_{87}), respectively. Note that, from a practical point of view, this algorithm allows P_{85} to conspire against its neighbours, but we cannot enforce the execution to be conspiracy-free.

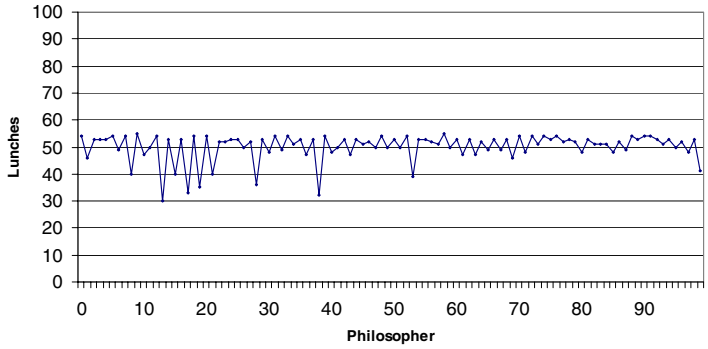
Figures 11(b) and (c) show the results we obtained when we ran the tests using Francez and Forman's proposal and the incremental one. These distributions are very similar because the latter proposal is an incremental version of the former which performs better, but does not attempt to produce a better distribution of lunches. The variation is, however, smaller than using the random proposal and it



(a)

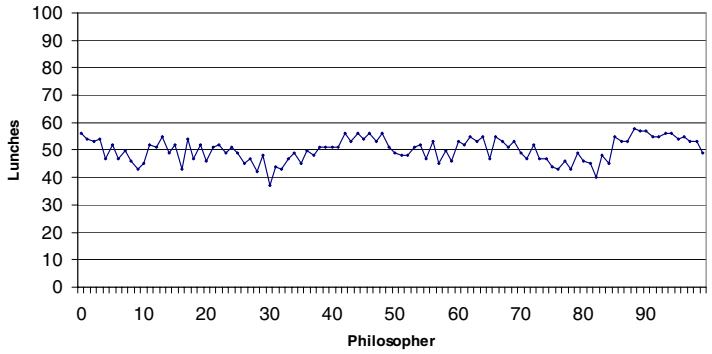


(b)

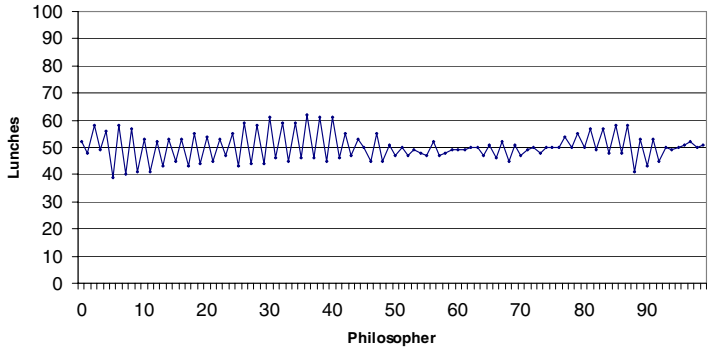


(c)

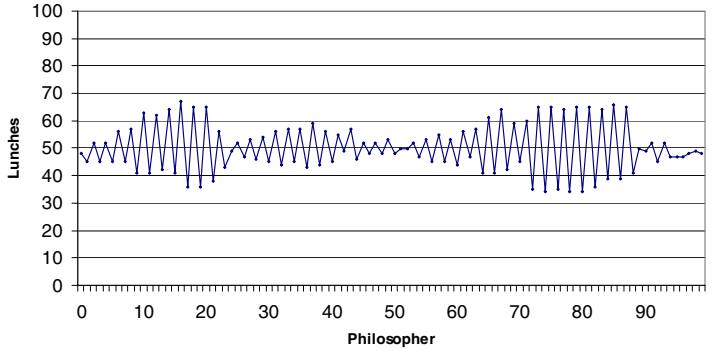
Figure 11. Distributions of lunches: (a) random proposal; (b) Francez and Forman's proposal and (c) incremental proposal; (d) $\mathcal{CFF}_k$ when $k = 1$ (small); (e) $\mathcal{CFF}_k$ when $k = 100$ (mild); (f) $\mathcal{CFF}_k$ when $k = 10\,000$ (large).



(d)



(e)



(f)

Figure 11. (Continued).

depends on the values produced by the random number generator used. In our implementation, we used the standard implementation provided by J# to produce numbers in the range [0 . . . 100]. None of the proposals, except for ours, can tune these variations.

Figures 11(d)–(f) show the distribution of lunches using $\mathcal{CFF}_k$ with several values of k ($k = 1, 100, 10\,000$). As the figures show, the variation of lunches amongst neighbouring philosophers is controlled by means of the value we assign to k. From this point of view, the main difference between our proposal and Francez and Forman's stems from the fact that the latter tends to select every interaction as many times as the others, independently of the number of philosophers, whereas our proposal can control the variation and the speed at which a system performs depending on the value we assign to k. This threshold can thus be viewed as a trade-off between efficiency and conspiracies.

7. CONCLUSIONS

Fairness has been researched by many authors in the context of $\mathcal{MI}$-based systems. They have devised several notions that attempt to avoid executions in which an interaction that is enabled sufficiently often is neglected. Strong fairness is quite an adequate notion, but it does not address fair finiteness and conspiracy problems.

In this article, we have shown that both problems may be addressed simultaneously by means of a new fairness notion that allows us to control to what extent conspiracies must be controlled and takes finite executions into account. We have defined our notion in the context of a framework we have designed and implemented to support $\mathcal{MI}$-based systems. The experimental results show that our notion can deal with conspiracies while still performing comparably to other proposals that do not attempt to solve this problem.

APPENDIX A. IP IN A NUTSHELL

Simple IP programs are of the following form:

$$\mathsf{S} :: [\mathsf{P}_1 \parallel \mathsf{P}_2 \parallel \ldots \parallel \mathsf{P}_\mathsf{n}],\ \textbf{where}$$
$$\mathsf{P}_1 :: \mathit{Body}_1$$
$$\mathsf{P}_2 :: \mathit{Body}_2$$
$$\ldots$$
$$\mathsf{P}_\mathsf{n} :: \mathit{Body}_\mathsf{n}$$

They model systems as collections of cooperating sequential processes whose relationships are based on multiparty interactions. Each process executes a body that is composed of a sequence of instructions.

Assignments. As usual, assignments are of the form $x := e$, where x denotes a local variable and e an expression over the local state of the process that executes this instruction. The null assignment is denoted as skip.

Interaction instructions. They are of the form $a\langle\overline{x := e}\rangle$, where a is the name of an interaction and $\overline{x := e}$ is an optional sequence of assignments referred to as the communication part since it allows a process to retrieve data from other processes. x refers to variables in the local

state of the process executing this instruction, but e may refer to variables in other processes participating in interaction a. Several improvements to this naive communication mechanism have been proposed, cf. [8,9,24].

Multi-choice instructions. They are of the form $[\Box_{i=1}^{n} G_i \rightarrow S_i]$, where each G_i is a guard and S_i is a list of instructions. Guards are of the form $B\&a\langle\overline{x := e}\rangle$, where B is a Boolean expression and the rest is an interaction instruction. They are passable, i.e. their corresponding instructions can be executed, iff the Boolean expression holds and interaction a is enabled. If $B\&$ is omitted, it is interpreted as $true\&$; if $\&a\langle\overline{x := e}\rangle$ is omitted, it is interpreted as $\&\langle\rangle$, where $\langle\rangle$ denotes an anonymous local interaction. Note that both parts of a guard cannot be omitted.

Multi-choice loops. They are of the form $*[\Box_{i=1}^{n} G_i \rightarrow S_i]$. Their semantics is similar to a multi-choice instruction, except for the fact that the whole instruction is repeated until none of the Boolean expressions that guard the alternatives is true.

APPENDIX B. PREVIOUS RESULTS

A preliminary version of this work was presented at the Euro-Par 2002 conference [30]. There, we presented a notion called $\mathcal{SKF}$ (*strong k-fairness*), which differs from $\mathcal{CFF}_k$ in that the set of linked interactions was calculated at runtime, whereas it is now calculated at compile time. That is, two interactions where considered to be linked as long as they had a common participant at runtime, not at compile time.

In spite of being so similar, the results are very different. Figure B1 shows that $\mathcal{CFF}_k$ alleviates conspiracies better than $\mathcal{SKF}$ because the variation is smaller. Since philosophers are continuously offering to participate in an interaction, be it a Get interaction or a Rel interaction, $\mathcal{CFF}_k$ introduces more delays because the set of interactions linked at compile time is usually greater than the set of interactions linked at runtime, and thus needs more interactions to be stable before selecting one of them, which allows us to control conspiracies better than $\mathcal{SKF}$.

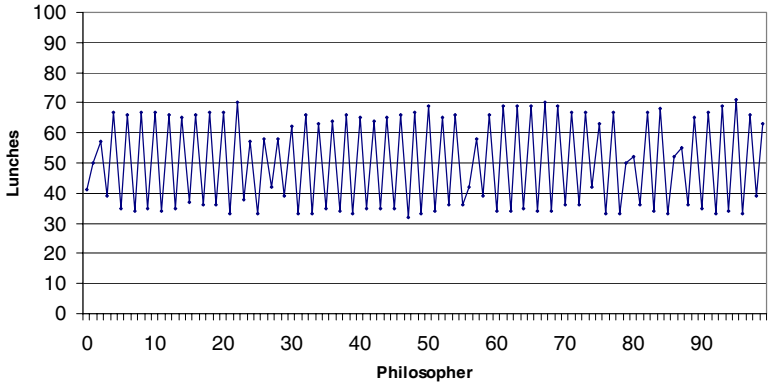


Figure B1. Distribution of lunches using $\mathcal{SKF}$ ($k = 1$).

ACKNOWLEDGEMENTS

We are thankful to our referees and Professor Mavronicolas for their insightful suggestions and their contributions to improve our results. We would also like to thank the participants to the Euro-Par 2002 Conference for engaging in fruitful discussion on fairness and $\mathcal{MI}$-based systems with us.

REFERENCES

1. Francez N, Forman I. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison-Wesley: Reading, MA, 1996.
2. Francez N. *Fairness*. Springer: Berlin, 1986.
3. Lamport L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers* (*Lecture Notes in Computer Science*, vol. 1845). Addison-Wesley: Boston, MA, 2002.
4. Alur R, Henzinger TA. Finitary fairness. *ACM Transactions on Programming Languages and Systems* 1998; **20**(6):1171–1194.
5. Attie PC, Francez N, Grumberg O. Fairness and hyperfairness in multiparty interactions. *Distributed Computing* 1993; **6**(4):245–254.
6. Beauquier J, Datta AK, Gradinariu M, Magniette F. Self-stabilizing local mutual exclusion and daemon refinement. *Proceedings of the DISC 2000 International Conference* (*Lecture Notes in Computer Science*, vol. 1914). Springer: Berlin, 2000; 223–237.
7. Jayasimha D, Dershowitz N. Bounded fairness. *Technical Report TR-615*, Center for Supercomputing Research and Development. University of Illinois, 1986.
8. Corchuelo R, Pérez JA, Ruiz–Cortés A. Aspect-oriented interaction in multi-organizational Web-based systems. *Computer Networks* 2003; **41**(4):385–406.
9. Corchuelo R, Pérez JA, Toro M. A multiparty coordination aspect language. *ACM Sigplan* 2000; **35**(12):24–32.
10. Joung YJ. A comprehensive study of the complexity of multiparty interaction. *Journal of the ACM* 1996; **43**(1):75–115.
11. Tang P, Muraoka Y. Parallel programming with interacting processes. *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, LCPC'99* (*Lecture Notes in Computer Science*, vol. 1863). Springer: Berlin, 2000; 201–218.
12. Felber P, Reiter MK. Advanced concurrency control in Java. *Concurrency and Computation: Practice and Experience* 2002; **14**(4):261–285.
13. Fayad M. E–Frame: A process-based, object-oriented framework for e-commerce. *Proceedings of the International Conference on Internet Computing IC'2001*. CSREA Press: Las Vegas, NV, 2001; 124–128.
14. Keen A, Ge T, Maris J, Olsson R. JR: Flexible distributed programming in an extended Java. *Proceedings 21st International Conference on Distributed Computing Systems, ICDCS'01*. IEEE Press: Los Alamitos, CA, 2001; 575–584.
15. Lea D. *Concurrent Programming Using Java: Design Principles and Pattern*. Addison-Wesley: Reading, MA, 1999.
16. Lynch NA, Merritt M, Weihl WE, Fekete A. *Atomic Transactions* (*Lecture Notes in Computer Science*, vol. 1845). Morgan Kaufmann: San Mateo, CA, 1994.
17. Kindler E, Walter R. Mutex needs fairness. *Information Processing Letters* 1997; **62**(1):31–39.
18. Schneider FB, Lamport L. Another position paper on 'fairness'. *Software Engineering Notes* 1988; **13**(3):1–2.
19. Dijkstra EW. Position paper on 'Fairness'. *Software Engineering Notes* 1988; **3**(2):18–20.
20. Best E. Fairness and conspiracies. *Information Processing Letters* 1984; **18**(3):215–220.
21. Best E. Erratum: Fairness and conspiracies. *Information Processing Letters* 1984; **19**(4):162.
22. Lamport L. Fairness and hyperfairness. *Distributed Computing* 2000; **13**(4):239–245.
23. Apt KR, Francez N, Katz S. Appraising fairness in languages for distributed programming. *Distributed Computing* 1988; **2**(4):226–241.
24. Corchuelo R. Prototyping constraint-based specifications of distributed systems. *PhD Thesis*, Facultad de Informática y Estadística, Dpto. de Lenguajes y Sistemas Informáticos, University of Sevilla, 1999.
25. Corchuelo R, Ruiz D, Toro M, Ruiz–Cortés A. Implementing multiparty interactions on a network computer. *Proceedings XXVth Euromicro Conference*. IEEE Press: Milan, Italy, 1999; 458–465.
26. Sarang PG, Adatia E, Jouhier B. $J^{\#}$. Wrox Press: Birmingham, U.K., 2002.
27. Olderog E, Apt KR. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages and Systems* 1988; **10**(3):420–255.
28. Joung YJ. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theoretical Computer Science* 2000; **243**(1–2):307–338.
29. Joung YJ. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems* 1998; **9**(2):137–149.
30. Ruiz D, Corchuelo R, Pérez JA, Toro M. An algorithm for ensuring fairness and liveness in non-deterministic systems based on multiparty interactions. *Proceedings of the Euro–Par 2002 International Conference* (*Lecture Notes in Computer Science*, vol. 2400). Springer: Berlin, 2002; 563–572.