An Algorithm for Ensuring Fairness and Liveness in Non-deterministic Systems Based on Multiparty Interactions*

David Ruiz, Rafael Corchuelo, José A. Pérez, and Miguel Toro

Universidad de Sevilla, E. T. S. Ingenieros Informáticos,
Av. de la Reina Mercedes s/n, Sevilla, E–41012, Spain
druiz@lsi.us.es, http://tdg.lsi.us.es

Abstract. Strong fairness is a notion we can use to ensure that an element that is enabled infinitely often in a non–deterministic programme, will eventually be selected for execution so that it can progress. Unfortunately, "eventually" is too weak to induce the intuitive idea of liveliness and leads to anomalies that are not desirable, namely *fair finiteness* and *conspiracies*. In this paper, we focus on non–deterministic programmes based on multiparty interactions and we present a new criteria for selecting interactions called *strong k–fairness* that improves on other proposals in that it addresses both anomalies simultaneously, and k may be set a priori to control its goodness. We also show our notion is feasible, and present an algorithm for scheduling interactions in a strongly k–fair manner using a theoretical framework to support the multiparty interaction model. Our algorithm does not require to transform the source code to the processes that compose the system; furthermore, it can deal with both terminating and non–terminating processes.

1 Introduction

Fairness is an important liveliness concept that becomes essential when the execution of a programme is non–deterministic [8]. This may be a result of the inherently non–deterministic constructs that the language we used to code it offers, or a result of the interleaving of atomic actions in a concurrent and/or distributed environment.

Intuitively, an execution of a programme is fair iff every element under consideration that is enabled sufficiently often is executed sufficiently often, which prevents undesirable executions in which an enabled element is neglected forever. The elements under consideration may range from alternatives in a non–deterministic multi–choice command to high–level business rules, and combined with a precise definition of "*sufficiently often*" lead to a rich lattice of fairness notions that do not collapse, i.e., are not equivalent each other [3].

* This article was supported by the Spanish Interministerial Commission on Science and Technology under grant TIC2000-1106-C02-01.

There is not a prevailing definition, but many researchers agree in that so called *strong fairness* deserves special attention [8] because it may induce termination or eventual response to an event. Technically, an execution is said to be strongly fair iff every element that is enabled indefinitely often is executed infinitely often, i.e., it prevents elements that are enabled infinitely often, but not necessarily permanently, from being neglected forever.

In this paper, we focus on concurrent and/or distributed programmes that use the multiparty interaction model as the sole means for process[1] synchronisation and communication. This interaction model is used in several academic programming languages like Scripts [8], Raddle [7] or IP [9] and in commercial programming environments like Microsoft .NET Orchestration [4] too. In this paper, we focus on IP because it is intended to have a dual role: on the one hand, it is intended to be a distributed system specification language equipped with sound semantics that turn it into a language amenable to formal reasoning, a rather important property; on the other hand, it is intended to be an assembler language supporting more sophisticated high–level specification languages such as LOTOS, ESTELLE, SDL [10] or CAL [5]. Next, we report on those issues, present some approaches to address them and give the reader a bird's–eye view of the rest of the paper.

1.1 Known Issues

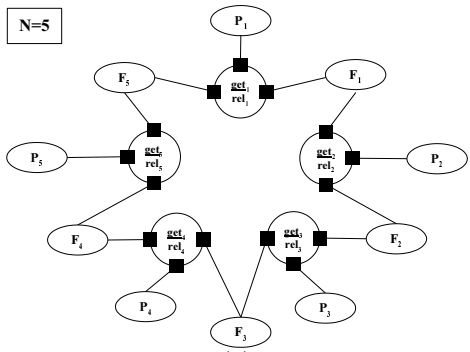
Figure 1 shows a solution to the well–known dining philosophers problem in IP. This classic multi-process synchronisation problem consists of five philosophers sitting at a table who do nothing but think and eat. There is a single fork between each philosopher, and they need to pick both forks up in order to eat. In addition, each philosopher should be able to eat as much as the rest, i.e., the whole process should be fair. This problem is the core of a large class of problems where a process needs to acquire a set of resources in mutual exclusion.

Get_i and Rel_i denote a number of three–party interactions that allow each philosopher P_i to get its corresponding forks F_i and $F_{i+1 \bmod N}$ in mutual exclusion with its neighbours ($i = 1, 2, \ldots, N$). For an interaction to become enabled, the set of processes that may eventually ready it, i.e., may eventually be willing to participate in the joint action it represents, need to be readying it simultaneously.

The only way to ensure that every philosopher that is hungry will eventually eat is by introducing a notion of fairness in the implementation of the language. However, strong fairness is not practical enough because of the following inherent problems:

Fair Finiteness: Every finite execution is strongly fair by definition. Figure 2.a shows a simple execution trace of an instantiation of the preceding programme in which $N = 5$. The notation $p.\chi$ means that process p readies the

[1] The term process refers to any autonomous, single–threaded computing artefact. It may be a process in an operating system, a thread, or even a hardware device.



$S :: [\|_{i=1}^{N} F_i \|\|_{i=1}^{N} T_i]$

$P_i :: *[\mathrm{Get}_i[] \rightarrow \mathit{eat}; \mathrm{Rel}_i[]; \mathit{think}]$

$F_i ::$
$*[\ \mathrm{Get}_i[] \rightarrow \mathrm{Rel}_i[]$
$\ [] \ \mathrm{Get}_{(i+1 \bmod N)}[] \rightarrow \mathrm{Rel}_{(i+1 \bmod N)}[]$
$]$

(b)

Fig. 1. A solution to the dining philosophers problem in IP.

set of interactions χ. Notice that for any finite n, this execution is technically strongly fair, despite Get_2 being enabled n times but never selected. If $n = 10$, this execution may be considered fair from an intuitive point of view, but if $n = 1000$ it is not so intuitive to consider this behaviour fair.

Conspiracies: It does not take into account conspiracies in which an interaction never gets enabled because of an unfortunate interleaving of independent atomic actions. For instance, the execution shown in Figure 2.b is strongly fair for any $n \geq 0$, but notice that due to an unfortunate interleaving, interaction Get_2 is never readied by all of its participants at the same time and thus never gets enabled.

The above problems show that strong fairness (and other notions that rely on infiniteness and eventuality) fails to capture the intuitive idea of inducing liveliness. Although it may be the only way to proof termination or eventual response to an event during an infinite execution, "eventual" is usually too weak for practical purposes because any practical running programme must necessarily stop or be stopped a day.

1.2 Related Work

These issues motivated several authors to research stronger notions. Here we focus on two approaches called *strong finitary fairness* [1] and *hyperfairness* [2].

An execution is *strongly finitarily fair* iff there exists a natural number k (not known a priori) such that every interaction that is enabled infinitely often is executed at least every k steps. Although this notion introduces additional liveliness because it bounds the number of times an enabled interaction may be neglected, it has several drawbacks: (i) k is not known a priori, and thus it cannot be used to fine-tune a potential scheduler depending on the nature of the system it is scheduling; (ii) it does not prevent unfair finiteness; (iii) it does not prevent conspiracies; and, to the best of our knowledge, (iv) no general algorithm implementing it has been produced. (The authors do only present a

$$P_1.\{Get_1\}, P_2.\{Get_2\},$$
$$(P_5.\{Get_5, Get_1\}, P_2.\{Get_2, Get_3\}, P_1.\{Get_1, Get_2\}, Get_1[],$$
$$P_1.\{Rel_1\}, P_1.\{Rel_1, Rel_2\}, P_5.\{Rel_5, Rel_1\}, Rel_1[], P_1.\{Get_1\})^n$$

(a)

$$P_1.\{Get_1\}, P_2.\{Get_2\}P_3.\{Get_3\},$$
$$(P_5.\{Get_5, Get_1\}, P_3.\{Get_3, Get_4\},$$
$$P_1.\{Get_1, Get_2\}, Get_1[], P_2.\{Get_2, Get_3\}, Get_3[], P_1.\{Rel_1\},$$
$$P_1.\{Rel1, Rel_2\}, P_5.\{Rel_5, Rel_1\}, Rel_1[], P_3.\{Rel_3\},$$
$$P_2.\{Rel_2, Rel_3\}, P_3.\{Rel_3, Rel_4\}, Rel_3[], P_1.\{Get_1\}, P_3.\{Get_3\})^n$$

(b)

Fig. 2. Strong fairness anomalies.

transformational approach suitable to be used in the context of Büchi automata [12,11].)

Hyperfairness also deserves attention because it alleviates the second problem. Technically, an execution is hyperfair iff it is finite or every interaction that may get enabled infinitely often, becomes enabled infinitely often. It is important to notice that this definition diverges from classical notions in that the latter imply eventual execution of an interaction if it gets enabled sufficiently often, whereas hyperfairness does only imply eventual enablement. Subsequent execution is under the criterion of an implied underlying classical fairness notion. Thus, this notion prevents conspiracies due to unfortunate interleaving of independent atomic actions but combined with finitary or strong fairness suffers from fair finiteness. To the best of our knowledge, no general algorithm able to implement hyperfairness has been produced. However, the authors presented a transformational approach by means of which we can transform an IP programme into an equivalent strongly hyperfair form, which implies modification of the source code and creation of explicit schedulers for each programme. This may be acceptable in the context of research languages, but it is not practical enough in real–world languages in which processes or components are available only in binary form and need to be scheduled without any knowledge of their internal details. Furthermore, it does not address the issue of fair finiteness.

1.3 Overview

In this paper, we present a new notion called *strong k–fairness* that solves fair finiteness and conspiracies. Intuitively, an execution is strongly k–fair iff no interaction is executed more than k times unless the set of interactions that share processes with it is stable, i.e., the processes that participate in them are waiting for interaction or finished, and it is the oldest in the group, i.e., the one that has not been executed for a longer period of time.

We present a theoretical interaction framework to formalize the multiparty interaction model. Furthermore, we present an algorithm that uses this frame-

work for scheduling interactions in a strongly k–fair manner, and it is not dependent on the internal details of the processes that compose the system, i.e., it is not a transformational approach.

The succeeding sections are organised as follows: Section 2 presents our theoretical interaction framework; Section 3 presents a formal definition of strong k–fairness; Section 4 describes a scheduler we can use to implement this notion; finally, Section 5 reports on our main conclusions.

2 A Theoretical Framework to Support the Multiparty Interaction Model

Next, we present a formal definition of our abstract interaction framework.

Definition 1 (Static Characterisation of a System) *A system Σ is a 2–tuple (P_Σ, I_Σ) in which $P_\Sigma \neq \emptyset$ is a finite set of autonomous processes and $I_\Sigma \neq \emptyset$ is a finite set of interactions. We denote the set of processes that may eventually ready interaction x as $\mathbb{P}(x)$ (participants of interaction x). A configuration is a mathematical object that may be viewed as a snapshot of a system at run time. We denote them as $C, C', C_1, C_2 \ldots$*

An event is a happening that induces a system to transit from a configuration to another. In our model, we take into account the following kinds of events: $p.\iota$, which indicates that process p executes an atomic action that does only involve its local data; $p.\chi$, which indicates that process p is readying the interactions in set χ (notice that when $\chi = \emptyset$, process p arrives at a fixed point that we may interpret as its termination); and x, which indicates that interaction x has been selected and the processes participating in it can execute the corresponding joint action atomically.

Definition 2 (Dynamic Characterisation of a System) *An execution of a system Σ is a 3–tuple (C_0, α, β) in which C_0 denotes its initial configuration, $\alpha = [C_1, C_2, C_3, \ldots]$ is a maximal (finite or infinite) sequence of configurations through which it proceeds, 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.*

We denote the rule that captures the underlying semantics that control the transition between configurations as $\longrightarrow\!\!\!\!\!\rightarrow_{\mathrm{L}}$. For instance, $C \xrightarrow{e}\!\!\!\!\!\rightarrow_{\mathrm{L}} C'$ 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[2]: $C_0 \xrightarrow{e_1}\!\!\!\!\!\rightarrow_{\mathrm{L}} C_1 \xrightarrow{e_2}\!\!\!\!\!\rightarrow_{\mathrm{L}} C_2 \xrightarrow{e_3}\!\!\!\!\!\rightarrow_{\mathrm{L}} \cdots$

[2] Notice that the exact formulation of $\longrightarrow\!\!\!\!\!\rightarrow_{\mathrm{L}}$ depends completely on the language in which the system under consideration was written.

Definition 3 (Static Characterisation of a Process) *Process p is waiting a interaction set Υ at the i–th configuration in execution λ iff it has arrived at a point in its execution in which executing any $x \in \Upsilon$ is one of its possible continuations. Process p is finished at the i–th configuration in execution λ iff it can neither execute any local computation nor any interaction.*

$$\begin{aligned} \text{Waiting}(\lambda, p, \Upsilon, i) &\iff \exists k \in [1..i] \cdot \beta(k) = p.\chi \wedge \Upsilon \subseteq \chi \wedge \nexists j \in [k+1..i] \cdot \beta(j) = x \wedge x \in \chi \\ \text{Finished}(\lambda, p, i) &\iff \exists k \in [1..i] \cdot \beta(k) = p.\emptyset \end{aligned} \tag{1}$$

Definition 4 (Static Characterisation of an Interaction) *Interaction x is enabled at the i–th configuration in execution λ iff all of the processes in $\mathbb{P}(x)$ are readying x at that configuration. Interaction x is stable at the i–th configuration in execution λ iff it is either enabled or disabled at that configuration.*

$$\begin{aligned} \text{Enabled}(\lambda, x, i) &\iff \forall p \in \mathbb{P}(x) \cdot \text{Waiting}(\lambda, p, \{x\}, i) \\ \text{Stable}(\lambda, x, i) &\iff \forall p \in \mathbb{P}(x) \cdot \exists \Upsilon \subseteq I_\Sigma \cdot \text{Waiting}(\lambda, p, \Upsilon, i) \end{aligned} \tag{2}$$

Definition 5 (Dynamic Characterisation of an Interaction) *The set of interactions linked to interaction x at the i–th configuration in execution λ is the set of interactions such that there exists a process that is readying x and any of those interactions simultaneously. We define the execution set of interaction x at the i–th configuration in execution λ as the set of indices up to i that denote the configurations at which interaction x has been executed.*

$$\begin{aligned} \text{Linked}(\lambda, x, i) &= \{y \in I_\Sigma \cdot \exists p \in P_\Sigma \cdot \text{Waiting}(\lambda, p, \{x, y\}, i)\} \\ \text{ExeSet}(\lambda, x, i) &= \{k \leq i \cdot \beta(k) = x\} \end{aligned} \tag{3}$$

3 Strong k–Fairness

Intuitively, an execution is strongly k–fair iff no interaction is executed more than k times unless all of the interactions that are linked to it when it is executed are stable and it is the oldest amongst them.

Definition 6 (Strongly k–Fair Execution) *Let $\lambda = (C_0, \alpha, \beta)$ be an execution of a system, and k a non–null natural number. λ is strongly k–fair iff predicate $\mathcal{SKF}(\lambda, k)$ holds.*

$$\begin{aligned} \mathcal{SKF}(\lambda, k) \iff \forall x \in I_\Sigma, i \in \text{ExeSet}(\lambda, x, \infty) \cdot \mathit{Enabled}(\lambda, x, i) \wedge \\ (\text{LStable}(\lambda, x, i) \wedge \text{LOldest}(\lambda, x, i) \vee \\ \neg\, \text{LStable}(\lambda, x, i) \wedge \Delta(\lambda, x, i) \leq k) \end{aligned} \tag{4}$$

This definition relies on a number of auxiliary predicates and functions we have introduced for the sake of simplicity. LStable is a predicate we use to determine if an interaction and those that are linked to it are stable at a given configuration in an execution. Its formal definition follows:

$$\text{LStable}(\lambda, x, i) \iff \forall y \in \text{Linked}(\lambda, x, i) \cup \{x\} \cdot \text{Stable}(\lambda, y, i) \tag{5}$$

LOldest is a predicate we use to determine if an interaction is older than any of the interactions to which it is linked or, in the worst case, is the same age. Its definition follows:

$$\mathrm{LOldest}(\lambda, x, i) \iff \forall y \in \mathrm{Linked}(\lambda, x, i) \cdot \mathrm{Age}(\lambda, x, i) \geq \mathrm{Age}(\lambda, y, i) \tag{6}$$

The age of an interaction is the number of configurations that have elapsed since it was executed for the last time, or ∞ if it has never been executed so far.

$$\mathrm{Age}(\lambda, x, i) = \begin{cases} i - \max \mathrm{ExeSet}(\lambda, x, i) & \text{if } \mathrm{ExeSet}(\lambda, x, i) \neq \emptyset \\ \infty & \text{otherwise} \end{cases} \tag{7}$$

Δ is a function that maps an event trace, an interaction and an index into the number of times it has executed in the presence of a non–empty set of linked interactions that was not stable. Its definition follows:

$$\Delta(\lambda, x, i) = \sum_{\phi \leq k < i} (\lambda_\beta(k) = x \wedge \mathrm{Linked}(\lambda, x, k) \neq \emptyset \wedge \neg\, \mathrm{LStable}(\lambda, x, k)) \tag{8}$$

where $\sum$ denotes the counter quantifier ($\sum_{a \in A} P(a) \triangleq |\{a \in A \cdot P(a)\}|$), and ϕ is defined as follows (notice that we denote the maximum of an empty set as $\perp$):

$$\phi \triangleq \begin{cases} j \text{ if } j = \max\{k \in \mathrm{ExeSet}(\lambda, x, i) \cdot \mathrm{LStable}(\lambda, x, k)\} \wedge j \neq \perp \\ 1 \text{ otherwise} \end{cases} \tag{9}$$

4 A Strongly k–Fair Scheduler

Our algorithm is based on previous proposals which want to resolve another problems in the context of the multiparty interactions [6,14,13].

The idea behind our algorithm for scheduling interactions in a strongly k–fair manner consists of arranging the set of interactions into a queue τ so that the closer an interaction is to the rear, the less time has elapsed since it was executed for the last time. Furthermore, each interaction has an associated counter δ we use to count how many times it has been semi–enabled in presence of linked interactions.

To know the state in which a process or an interaction is, we use a map φ from the set of interactions into the set of processes that are readying it. We update it each time the selection module detects a transition $\xrightarrow{p.X}_{\mathrm{L}}$ occurs, or an interaction is executed. Our algorithm selects for execution interaction x as long as it is the first enabled one in queue τ, and the set of interactions linked to it is stable.

We describe the operational semantics of our strong k–fairness algorithm using transition rule $\longrightarrow_{\mathcal{SKF}}$ on configurations $D, D', D_1, \ldots$. These configurations are composed of the configuration C of the programme and the data structures we need to select interactions. Next, we define theses data structures and the functions that allows us to update it.

The extended configurations on which our algorithm works are of the form $(\tau, \varphi, \delta, \vartheta)$, where τ is an interaction queue, φ is a readiness map, δ is a semi–enablement map, and ϑ denotes the set of processes that are finished.

Definition 7 (Data Structures) *φ denotes a map from interactions into sets of processes. $\varphi(x)$ denotes the set of processes that are readying x. $\vartheta \subseteq P_\Sigma$ denotes the set of processes that are finished, i.e., can neither execute local computations nor ready any interaction. δ is a map from the set of interactions into the set of natural numbers. $\delta(x)$ denotes the number of times any interaction linked to x has been executed while x was semi–enabled. The higher $\delta(x)$ is, the higher the probability of conspiracy is. τ denotes a queue in which the set of interactions has been arranged so that the closer they are to the rear, the less time has elapsed since they were executed for the last time. As usual, we consider a queue of interactions is a map from a subset of natural numbers into the set of interactions.*

The initial extended configuration of our algorithm is of the form $(\tau_0, \varphi_0, \delta_0, \vartheta_0)$. It does not matter the order in which interactions are initially arranged into τ, but φ_0 must satisfy that $\forall x \in \operatorname{dom}\varphi_0 \cdot \varphi_0(x) = \emptyset$, δ_0 must satisfy that $\forall x \in \operatorname{dom}\delta_0 \cdot \delta_0(x) = 0$, and $\vartheta_0 = \emptyset$.

Definition 8 (Functions) *When process p readies a set of interactions χ, we use function* $\mathrm{AddOffer}(\varphi, p, \chi)$ *to update map φ, and when interaction x is executed, we use function* $\mathrm{RemoveOffer}(\varphi, x)$*. By definition, process p finishes when it readies an empty set of interactions, and function* $\mathrm{AddFinished}(\vartheta, p)$ *updates map ϑ. When interaction x is selected for execution, we use function Order to move it to the rear of τ, not necessarily to the last position. When interaction x is selected for execution, we use function* $\mathrm{Update}(\varphi, \delta, x)$ *to create a new semi–enablement. Predicate* $\mathrm{Stabilised}(\Upsilon, \varphi, \vartheta)$ *holds iff all of the processes that may eventually ready an interaction in Υ has readied it or are finished.* $\mathrm{EnblDisj}(\tau, \varphi)$ *denotes the set of interactions that are enabled and it does not exists any preceding interaction in τ that is enabled and linked to them.*

$$
\begin{aligned}
\mathrm{AddOffer}(\varphi, p, \chi) =& \{x \mapsto \varphi(x) \cdot x \in \operatorname{dom}\varphi \wedge x \notin \chi\} \cup \\
& \{x \mapsto \varphi(x) \cup \{p\} \cdot x \in \operatorname{dom}\varphi \wedge x \in \chi\} \\
\mathrm{RemoveOffer}(\varphi, x) =& \{x \mapsto \varphi(x) \setminus \mathbb{P}(x) \cdot x \in \operatorname{dom}\varphi\} \\
\mathrm{AddFinished}(\vartheta, p) =& \begin{cases} \vartheta \cup \{p\} & \text{if } \chi \neq \emptyset \\ \vartheta & \text{if } \chi = \emptyset \end{cases} \\
\mathrm{Order}(\tau, \delta) =& \tau' \Leftrightarrow \operatorname{dom}\tau = \operatorname{dom}\tau' \wedge \operatorname{ran}\tau = \operatorname{ran}\tau' \wedge \\
& \forall x_1, x_2 \in \operatorname{ran}\tau \cdot (\tau'^{-1}(x_1) \leq \tau'^{-1}(x_2) \Rightarrow \delta(x_1) \geq \delta(x_2)) \\
\mathrm{Update}(\varphi, \delta, x) =& \delta \otimes \{x \mapsto 0\} \otimes \{y \mapsto \delta(y) + 1 \cdot y \in \mathcal{S} \setminus \{x\} \wedge \neg\, \mathrm{Stabilised}(\mathcal{S}, \varphi, \vartheta)\} \\
\mathrm{Stabilised}(\Upsilon, \varphi, \vartheta) \Leftrightarrow& \forall x \in \Upsilon \cdot \forall p \in \mathbb{P}(x) \cdot p \in \operatorname{ran}\varphi \vee p \in \vartheta \\
\mathrm{EnblDisj}(\tau, \varphi) =& \{x \in \operatorname{dom}\varphi \cdot \mathbb{P}(x) = \varphi(x) \wedge \nexists y \in \mathcal{S} \cdot \mathbb{P}(y) = \varphi(y) \wedge \tau^{-1}(y) < \tau^{-1}(x)\}
\end{aligned}
\tag{10}
$$

where $\mathcal{S} \triangleq \{z \in \operatorname{dom}\varphi \cdot \varphi(z) \cap \varphi(x) \neq \emptyset\}$.

Our algorithm is formally defined by means of the inference rules presented in Figure 3. Rule 11 is straightforward because it describes how the data structures are updated each time process p readies a set of interactions χ. Rule 12 describes which interaction must be selected so that the execution is strongly k–fair. The antecedent is complex, but the reasoning behind it is quite simple. Assume x is an enabled interaction and there is not a conflicting enabled interaction before x in queue τ, i.e., $x \in \text{EnblDisj}(\tau, \varphi)$; in this context x is selected for execution iff any of the following three conditions hold:

$$\frac{C \xrightarrow{p \cdot \chi}_{\text{L}} C' \wedge \varphi' = \text{AddOffer}(\varphi, p, \chi) \wedge \vartheta' = \text{AddFinished}(\vartheta, p)}{(C, \tau, \varphi, \delta, \vartheta) \xrightarrow{p \cdot \chi}_{\mathcal{SKF}} (C', \tau, \varphi' \delta, \vartheta')} \tag{11}$$

$$\frac{\begin{array}{c} x \in \text{EnblDisj}(\tau, \varphi) \wedge \mathcal{S} = \{z \in \text{dom}\, \varphi \cdot (\varphi(z) \cap \varphi(x) \neq \emptyset)\} \wedge \\ \tau' = \text{Order}(\tau, \delta') \wedge \varphi' = \text{RemoveOffer}(\varphi, x) \wedge \delta' = \text{Update}(\varphi, \delta, x) \wedge \\ (\mathcal{S} = \{x\} \vee \text{Stabilised}(\mathcal{S}, \varphi, \vartheta) \vee (\mathcal{S} \neq \{x\} \wedge \max_{y \in \mathcal{S} \setminus \{x\}} \delta(y) < k)) \end{array}}{(C, \tau, \varphi, \delta, \vartheta) \xrightarrow{x}_{\mathcal{SKF}} (C', \tau', \varphi', \delta', \vartheta) \wedge C \xrightarrow{x}_{\text{L}} C'} \tag{12}$$

Fig. 3. Algorithm for strongly k–fair scheduler.

1. It is not conflicting with other interactions ($\mathcal{S} = \{x\}$).
2. The interactions which it is conflicting with are stabilised, i.e., all of their participants are readying it or finished ($\text{Stabilised}(\mathcal{S}, \varphi, \vartheta)$).
3. The semi–enablement counter associated with each conflicting interaction is less than k ($\mathcal{S} \neq \{x\} \wedge \max_{y \in \mathcal{S} \setminus \{x\}} \delta(y) < k$).

5 Conclusions and Future Work

In this paper, we have presented strong k–fairness in the context of concurrent and/or distributed programmes in which multiparty interaction models are the sole mean for process synchronisation and communication. An important contribution is the concept of semi–enablement that we use to forecast conspiracies and solve them. k can thus be viewed as a semi–enablement threshold that characterises the goodness of our notion because it makes our selection criterion more or less demanding. If k is minimum, the pace at which conflicting interactions are executed depends then on the participant that spends more time at doing local computations because, in this case, no interaction can be selected for execution unless the set of potentially conflicting interactions is consolidated. If k is very distant from its minimum, the algorithm introduces little delay because interactions may be scheduled as soon as they are enabled. The exact choice of k depends on the features of the programme under consideration and can only be tuned by means of experimentation.

In the future, we are going to research on a new algorithm in which k can be adapted at run–time. The idea is to record information about the execution of a programme so as to be able to forecast if an interaction is suffering from conspirations. Thus we might be able to introduce delays only when the forecast predicts it is necessary. Data mining techniques seem to be promising to forecast and will be paid much attention.

References

1. R. Alur and T. A. Henzinger. Finitary fairness. *ACM Transactions on Programming Languages and Systems*, 20(6):1171–1194, November 1998.
2. P.C. Attie, N. Francez, and O. Grumberg. Fairness and hyperfairness in multiparty interactions. *Distributed Computing*, 6(4):245–254, 1993.
3. E. Best. *Semantics of Sequential and Parallel Programs*. Prentice Hall, New York, 1996.
4. J. Conard. *Introducing .NET*. Wrox Press, Inc, 2001.
5. R. Corchuelo, J.A. Pérez, and M. Toro. A multiparty coordination aspect language. *ACM Sigplan*, 35(12):24–32, December 2000.
6. R. Corchuelo, D. Ruiz, M. Toro, and A. Ruiz. Implementing multiparty interactions on a network computer. In *Proceedings of the XXVth Euromicro Conference (Workshop on Network Computing)*, Milan, September 1999. IEEE Press.
7. M. Evangelist, V.Y. Shen, I.R. Forman, and M. Graf. Using Raddle to design distributed systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 102–115. IEEE Computer Society Press, April 1988.
8. N. Francez. *Fairness*. Springer–Verlag, 1986.
9. N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.
10. D. Hogrefe. *Estelle, Lotos and SDL*. Springer–Verlag, Berlin, 1989.
11. J.R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proceedings of the 1960 International Congress of Logic, Methodology and Philosophy of Science*, pages 1–12. Stanford University Press, 1960.
12. E. Olderog and K.R. Apt. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages and Systems*, 10(3):420–455, July 1988.
13. J. A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An order-based, distributed algorithm for implementing multiparty interactions. In *Fifth International Conference on Coordination Models and Languages COORDINATION 2002*, pages 250–257, York, UK, 2002. Springer–Verlag.
14. J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An enablement detection algorithm for open multiparty interactions. In *ACM Symposium on Applied Computing SAC'02*, pages 378–384, Madrid, Spain, 2002. Springer–Verlag.