Time and Fairness in a Process Algebra with Non-Blocking Reading

Flavio Corradini[1], Maria Rita Di Berardini[1], and Walter Vogler[2]

[1] Dipartimento di Matematica e Informatica, Università di Camerino
{flavio.corradini, mariarita.diberardini}@unicam.it
[2] Institut für Informatik, Universität Augsburg
vogler@informatik.uni-augsburg.de

Abstract. We introduce the first process algebra with non-blocking reading actions for modelling concurrent asynchronous systems. We study the impact this new kind of actions has on fairness, liveness and the timing of systems, using as application Dekker's mutual exclusion algorithm we already considered in [4]. Regarding some actions as reading, this algorithm satisfies MUTEX liveness already under the assumption of fairness of actions. We demonstrate an interesting correspondence between liveness and the catastrophic cycles that we introduced in [6] when studying the performance of pipelining. Finally, our previous result on the correspondence between timing and fairness [4] scales up to the extended language.

1 Introduction

Read arcs are an extension of classical Petri nets to model non-destructive reading operations; they allow multiple concurrent reading of the same resource, a quite frequent situation in many distributed systems. Read-arcs represent *positive context conditions*, i.e. elements which are needed for an event to occur, but are not affected by it. As argued in [11], the importance of such elements is twofold. Firstly, they allow a faithful representation of systems where the notion of "reading without consuming" is commonly used, like database systems, concurrent constraint programming, or any computation framework based on shared memory. Secondly, they allow to specify directly and naturally a level of concurrency greater than in classical nets: two transitions reading the same place may occur in any order and also simultaneously; in classical nets, the transitions would be connected to the place by loops such that they cannot occur simultaneously. Read arcs have been used to model a variety of applications such as transaction serialisability in databases [14], concurrent constraint programming [12], asynchronous systems [15], and cryptographic protocols [9]. Semantics and expressivity of read arcs have been studied e.g. in the following: [2] discusses a step semantics. [1] discusses the expressiveness of timed Petri nets and timed automata and shows that timed Petri nets with read-arcs unify timed Petri nets and timed automata. Finally, [15] shows that read arcs add relevant expressivity; the MUTEX problem can be solved with nets having read arcs but not with ordinary nets having no read arcs.

In this paper we introduce the first process algebra with non-blocking reading; we add reading in the form of a read-action-prefix to PAFAS [7], a process algebra for modelling timed concurrent asynchronous systems. In [5] we provide two different ways to enhance our process algebra with such non-blocking actions: one is more flexible, but needs a two-level transition relation; here, due to lack of space, we only present the other, simpler one. Further details can be found in [5] where we also prove that each so-called read-proper process (cf. Section 2) in our setting can be translated to a process in the alternative approach with isomorphic behaviour.

Non-blocking actions have a direct impact on timed behaviour. Consider a system composed of two processes that read the same variable to prepare an output they produce together, modelled as $(r.o.\mathsf{nil} \parallel_{\{o\}} r.o.\mathsf{nil}) \parallel_{\{r,w\}} \mathsf{rec}\ x.\,(r.x + w.x)$ in PAFAS: $\mathsf{rec}\ x.\,(r.x + w.x)$ models the variable that can repeatedly be read with r or written with w (abstracting from values; $r.o.\mathsf{nil}$ models one process that performs r (together with the variable) and then o, synchronising with the other process. According to the PAFAS semantics, enabled actions are performed immediately or become urgent after one time unit and must occur then; thus, after at most one time unit, the first r occurs. Every time the variable performs action r, a new instance of r is generated that can let one time unit pass; thus, a second time unit might pass before the second r, and the output is produced within three time units in the worst-case. If the read action r were modelled as a non-blocking action, written $\mathsf{rec}\ x.\{r\} \triangleright w.x$, then the worst-case efficiency for producing the output would be two time units. In this case, after one time unit and the occurrence of the first r, the variable offers the same non-blocking action r, which is still urgent and has to be performed before the second time step.

In previous work, we have shown that our notion of time is in strong correspondence to (weak) fairness [4], which requires that an action (or a component) has to be performed whenever it is enabled continuously in a run. We have proven that each everlasting (or non-Zeno) timed process execution is fair and vice versa, where fairness is defined in an intuitive but complicated way in the spirit of [8]. We used these characterizations in [3] to study the liveness property for Dekker's mutual exclusion algorithm, and proved that Dekker is *live under* the assumption of *fairness of components but not under* the assumption of *fairness of actions*. One important result is that the extension with non-blocking actions preserves the above correspondence result between fairness of actions and timing. Another main result is that, in the new setting, Dekker's algorithm is live when assuming fairness of actions, provided we regard as non-blocking the reading of a variable (as r in $\mathsf{rec}\ x.\{r\} \triangleright w.x$ above) as well as its writing in the case that the written value equals the current value. This kind of re-write does not change the state of the variable and, hence, can be thought of as a non-destructive or non-consuming operation (allowing potential concurrent behaviour). This way of accessing a variable is not new and Remark 1 in Section 3 describes how it is implemented in the area of databases.

Finally, we develop an interesting connection between liveness of MUTEX algorithms and catastrophic cycles: we considered the latter in [6] studying the

very different problem of asymptotic performance for the specific, but often occurring class of request-response processes (having only actions *in* and *out*). A cycle in a transition system is *catastrophic* if it only contains internal actions and at least one time step, and we showed that a process can refuse to serve some request within finite time if and only if a reduced transition system of the process contains a catastrophic cycle. We also pointed out that the existence of catastrophic cycles in a reduced transition system can be determined in polynomial time. In the present paper, we show how to modify the process *Dekker* such that *Dekker* satisfies liveness assuming fairness of actions if and only if the modified process does not have a catastrophic cycle. This opens the way to check automatically the liveness property for MUTEX algorithms and, indeed, we have developed a tool for verifying this property.

The rest of the paper is organized as follows. The next section introduces PAFAS with read-prefixes, and its functional and temporal operational semantics. Section 3 introduces fairness of actions and relates it to timing. Finally, Section 4 investigates the liveness of Dekker's algorithm under the assumption of fairness of actions and presents the interesting connection between liveness and catastrophic cycles.

2 A process algebra for describing read behaviours

PAFAS [7] is a CCS-like process description language [10] (with a *TCSP*-like parallel composition), where basic actions are atomic and instantaneous but have associated an upper time bound (either 0 or 1, for simplicity); this can be used for evaluating the performance of asynchronous systems (but does not influence *functionality*, i.e. which actions are performed). Here, we extend PAFAS with the new operator $\triangleright$ to represent non-blocking behaviour of processes. Intuitively, $\{\alpha_1, \ldots, \alpha_n\}\triangleright P$ models a process like a variable or a more complex data structure that behaves as P but can additionally be read with $\alpha_1, \ldots, \alpha_n$: since being read does not change the state, actions $\alpha_1, \ldots, \alpha_n$ can be performed repeatedly without blocking a synchronization partner as described below.

We use the following notation: $\mathbb{A}$ is an infinite set of *basic actions*; the additional action τ represents internal activity, unobservable for other components, and $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau\}$. Elements of $\mathbb{A}$ are denoted by $a, b, c, \ldots$ and those of $\mathbb{A}_\tau$ by $\alpha, \beta, \ldots$. Actions in $\mathbb{A}_\tau$ can let time 1 pass before their execution, i.e. 1 is their maximal delay. After that time, they become *urgent* actions written $\underline{a}$ or $\underline{\tau}$; these have maximal delay 0. The set of urgent actions is denoted by $\underline{\mathbb{A}}_\tau = \{\underline{a} \mid a \in \mathbb{A}\} \cup \{\underline{\tau}\}$ and is ranged over by $\underline{\alpha}, \underline{\beta}, \ldots$. Elements of $\mathbb{A}_\tau \cup \underline{\mathbb{A}}_\tau$ are ranged over by μ and ν. We also assume that, for any $\alpha \in \mathbb{A}_\tau$, $\underline{\underline{\alpha}} = \underline{\alpha}$.

$\mathcal{X}$ (ranged over by $x, y, z, \ldots$) is the set of process variables, used for recursive definitions. $\Phi : \mathbb{A}_\tau \to \mathbb{A}_\tau$ is a *general relabelling function* if the set $\{\alpha \in \mathbb{A}_\tau \mid \emptyset \neq \Phi^{-1}(\alpha) \neq \{\alpha\}\}$ is finite and $\Phi(\tau) = \tau$. Such a function can also be used to define *hiding*: P/A, where the actions in A are made internal, is the same as $P[\Phi_A]$, where the relabelling function Φ_A is defined by $\Phi_A(\alpha) = \tau$ if $\alpha \in A$ and $\Phi_A(\alpha) = \alpha$ if $\alpha \notin A$.

In the following definition, initial processes are just processes of a standard process algebra extended with $\triangleright$. General processes are those reachable from the initial ones according to the operational semantics.

Definition 1. (*timed process terms*) The set $\tilde{\mathbb{P}}_1$ of *initial (timed) process terms* is generated by the following grammar

$$P ::= \mathsf{nil} \mid x \mid \alpha.P \mid \{\alpha_1, \ldots, \alpha_n\} \triangleright P \mid P + P \mid P \parallel_A P \mid P[\Phi] \mid \mathsf{rec}\, x.P$$

where nil is a constant, $x \in \mathcal{X}$, $\alpha \in \mathbb{A}_\tau$, $\{\alpha_1, \ldots, \alpha_n\}$ is a finite, nonempty subset of $\mathbb{A}_\tau$, Φ is a general relabelling function and $A \subseteq \mathbb{A}$ possibly infinite. We assume that recursion is both action-guarded and read-guarded (see below). The set $\tilde{\mathbb{P}}$ of (general) *(timed) process terms* is generated by the following grammar:

$$Q ::= P \mid \underline{\alpha}.P \mid \{\mu_1, \ldots, \mu_n\} \triangleright Q \mid Q + Q \mid Q \parallel_A Q \mid Q[\Phi] \mid \mathsf{rec}\, x.Q$$

where $P \in \tilde{\mathbb{P}}_1$ and $\{\mu_1, \ldots, \mu_n\} \subseteq \mathbb{A}_\tau \cup \underline{\mathbb{A}}_\tau$ (finite and nonempty) is a *read-set*, i.e. it does not contain α *and* $\underline{\alpha}$ for any $\alpha \in \mathbb{A}_\tau$. Terms not satisfying this property are not reachable from initial ones anyway (see Section 2.2). A variable $x \in \mathcal{X}$ is *action-guarded* in Q if it only appears in Q within the scope of a prefix $\mu.()$ with $\mu \in \mathbb{A}_\tau \cup \underline{\mathbb{A}}_\tau$. A term Q is *action-guarded* if each occurrence of a variable is guarded in this sense. Moreover, a variable $x \in \mathcal{X}$ is said to be *read-guarded* in Q if, for each subterm of Q of the form $\{\mu_1, \ldots, \mu_n\} \triangleright Q_1$, x is action-guarded in Q_1. We assume that recursion is action- and read-guarded, i.e. for each term $\mathsf{rec}\, x.Q$, the variable x is both action-guarded and read-guarded in Q. A process term is *closed* if every variable x is bound by the corresponding $\mathsf{rec}\, x$-operator; the set of closed timed process terms in $\tilde{\mathbb{P}}$ and $\tilde{\mathbb{P}}_1$, simply called *processes* and *initial processes* resp., is denoted by $\mathbb{P}$ and $\mathbb{P}_1$ resp.

The operators have their usual intuition, e.g. Q_1 and Q_2 run in parallel in $Q_1 \| _A Q_2$ and have to synchronize on all actions from A; we also use equations to define recursive processes. The essential point about the read-set operator is that $\{\mu_1, \ldots, \mu_n\} \triangleright Q$ can perform the actions from $\{\mu_1, \ldots, \mu_n\}$ without changing state (including urgencies and, hence, the syntax of the term itself), and the actions of Q in the same way as Q, i.e. the read-set is removed after such an action.

In the following, we will focus on the so-called read-proper terms, which do have a reasonable semantics. Read-proper terms only have "properly" nested reading behaviour, i.e. we want to avoid terms like $\{a\} \triangleright \{b\} \triangleright Q$ or $\{a\} \triangleright Q' + \{b\} \triangleright Q$; these terms violate the intuition for reading, see below. More formally, a term $Q \in \tilde{\mathbb{P}}$ is *read-guarded* if every subterm of Q of the form $\{\mu_1, \ldots, \mu_n\} \triangleright Q'$ is in the scope of some μ (i.e. in some subterm $\mu.()$). A term $Q \in \tilde{\mathbb{P}}$ is *read-proper* if each subterm $Q_1 + Q_2$ is read-guarded and, for each subterm $\{\mu_1, \ldots, \mu_n\} \triangleright Q_1$, Q_1 is read-guarded. With this definition, neither $\{a\} \triangleright \{b\} \triangleright Q$ nor $\{a\} \triangleright Q' + \{b\} \triangleright Q$ are read-proper, since the subterm $\{b\} \triangleright Q$ is not in the scope of some μ and, thus, also not read-guarded.

In the following preliminary definitions, set A represents the actions restricted by the environment (i.e. in a parallel context); $\mathcal{U}(\{\mu_1, \ldots, \mu_n\})$ denotes the set of urgent actions in $\{\mu_1, \ldots, \mu_n\}$.

Definition 2. (*urgent basic actions*) Let $Q \in \tilde{\mathbb{P}}$ and $A \subseteq \mathbb{A}_\tau$. The set $\mathcal{U}(Q, A)$ is defined by induction on Q. The *urgent actions* of Q are defined as $\mathcal{U}(Q, \emptyset)$ which we abbreviate to $\mathcal{U}(Q)$.

Nil, Var: $\mathcal{U}(\mathsf{nil}, A) = \mathcal{U}(x, A) = \emptyset$

Pref: $\mathcal{U}(\mu.P, A) = \begin{cases} \{\alpha\} & \text{if } \mu = \underline{\alpha} \text{ and } \alpha \notin A \\ \emptyset & \text{otherwise} \end{cases}$

Read: $\mathcal{U}(\{\mu_1, \ldots, \mu_n\} \triangleright Q, A) = \mathcal{U}(\{\mu_1, \ldots, \mu_n\}) \setminus A \cup \mathcal{U}(Q, A)$

Sum: $\mathcal{U}(Q_1 + Q_2, A) = \mathcal{U}(Q_1, A) \cup \mathcal{U}(Q_2, A)$

Par: $\mathcal{U}(Q_1 \parallel_B Q_2, A) = \bigcup_{i=1,2} \mathcal{U}(Q_i, A \cup B) \cup (\mathcal{U}(Q_1, A) \cap \mathcal{U}(Q_2, A) \cap B)$

Rel : $\mathcal{U}(Q[\Phi], A) = \Phi(\mathcal{U}(Q, \Phi^{-1}(A)))$

Rec: $\mathcal{U}(\mathsf{rec}\, x.Q, A) = \mathcal{U}(Q, A)$

Since the environment restricts the actions in A, $\mathcal{U}(\mu.P, A) = \{\alpha\}$ only if $\mu = \underline{\alpha}$ and $\alpha \notin A$; otherwise $\mathcal{U}(\mu.P, A) = \emptyset$; observe that an initial process P cannot have any urgent actions. The essential idea for parallel composition is that a synchronised action can be delayed if at least one component can delay it: $\mathcal{U}(Q_1 \parallel_B Q_2, A)$ includes the actions that are urgent in Q_1 or Q_2 when the actions in A and in B (the synchronising ones) are prevented, and the actions in B, but not in A, that are urgent both in Q_1 and in Q_2. The other rules are as expected. To keep the definition more intuitive, please note that $\mathcal{U}(Q, A) = \mathcal{U}(Q) \backslash A$.

The operational semantics exploits two functions on process terms: $\mathsf{clean}(\_)$ and $\mathsf{unmark}(\_)$. Function $\mathsf{unmark}(\_)$ simply removes all urgencies (inactive or not) in a process term $Q \in \tilde{\mathbb{P}}$. Function $\mathsf{clean}(\_)$ removes *all inactive urgencies* in a process term $Q \in \tilde{\mathbb{P}}$; when a process evolves and a synchronised action is no longer urgent or enabled in some synchronisation partner, then it should also lose its urgency in the other. Below, A in $\mathsf{clean}(Q, A)$ denotes the set of actions that are not enabled or urgent due to restrictions of the environment. For a read-set $\{\mu_1, \ldots, \mu_n\} \subseteq \mathbb{A}_\tau \cup \underline{\mathbb{A}}_\tau$, $\mathsf{clean}(\{\mu_1, \ldots, \mu_n\}, A)$ denotes the set we obtain by replacing $\underline{\alpha}$ in $\{\mu_1, \ldots, \mu_n\}$ by α whenever $\alpha \in A$.

Definition 3. (*cleaning inactive urgencies*) For any $Q \in \tilde{\mathbb{P}}$ we define $\mathsf{clean}(Q)$ as $\mathsf{clean}(Q, \emptyset)$ where, for $A \subseteq \mathbb{A}$, $\mathsf{clean}(Q, A)$ is defined below.

Nil, Var: $\mathsf{clean}(\mathsf{nil}, A) = \mathsf{nil}, \qquad \mathsf{clean}(x, A) = x$

Pref: $\mathsf{clean}(\mu.P, A) = \begin{cases} \alpha.P & \text{if } \mu = \underline{\alpha} \text{ and } \alpha \in A \\ \mu.P & \text{otherwise} \end{cases}$

Read: $\mathsf{clean}(\{\mu_1, \ldots, \mu_n\} \triangleright Q, A) = \mathsf{clean}(\{\mu_1, \ldots, \mu_n\}, A) \triangleright \mathsf{clean}(Q, A)$

Sum: $\mathsf{clean}(Q_1 + Q_2, A) = \mathsf{clean}(Q_1, A) + \mathsf{clean}(Q_2, A)$

Par: $\mathsf{clean}(Q_1 \parallel_B Q_2, A) = \mathsf{clean}(Q_1, A_1) \parallel_B \mathsf{clean}(Q_2, A_2)$
where $A_1 = A \cup (B \backslash \mathcal{U}(Q_2))$ and $A_2 = A \cup (B \backslash \mathcal{U}(Q_1))$

Rel: $\mathsf{clean}(Q[\Phi], A) = \mathsf{clean}(Q, \Phi^{-1}(A))[\Phi]$

Rec: $\mathsf{clean}(\mathsf{rec}\, x.Q, A) = \mathsf{rec}\, x.\, \mathsf{clean}(Q, A)$

2.1 The functional behaviour of PAFAS$_s$ processes

The transitional semantics describing the functional behaviour of PAFAS$_s$ processes indicates which basic actions they can perform.

Definition 4. (*Functional operational semantics*) Let $Q \in \tilde{\mathbb{P}}$ and $\alpha \in \mathbb{A}_\tau$. The SOS-rules defining the transition relation $\xrightarrow{\alpha} \subseteq (\tilde{\mathbb{P}} \times \tilde{\mathbb{P}})$ are given in Table 1. As usual, we write $Q \xrightarrow{\alpha} Q'$ if $(Q, Q') \in \xrightarrow{\alpha}$ and $Q \xrightarrow{\alpha}$ if there exists a $Q' \in \tilde{\mathbb{P}}$ such that $(Q, Q') \in \xrightarrow{\alpha}$. Similar conventions are applied for Definition 6.

Rules in Table 1 are quite standard (apart from using clean in the PAR-rules as explained above). Notice that timing can be disregarded in PREF: when an action is performed, one cannot see whether it was urgent or not, and thus $\underline{\alpha}.P \xrightarrow{\alpha} P$; furthermore, component $\alpha.P$ has to act *within* time 1, i.e. it can also act immediately, giving $\alpha.P \xrightarrow{\alpha} P$. Rules READ$_1$ and READ$_2$ say that $\{\mu_1, \ldots, \mu_n\} \triangleright Q$ can either repeatedly perform one of its non-blocking actions or evolve as Q. The use of the unmark in rule REC has no effects for an initial process, where REC is the standard SOS rule. For non-initial Q, we will explain this rule in Example 1. Symmetric rules have been omitted.

$$\textsc{Pref}\ \frac{\mu \in \{\alpha, \underline{\alpha}\}}{\mu.P \xrightarrow{\alpha} P} \qquad \textsc{Sum}\ \frac{Q_1 \xrightarrow{\alpha} Q'}{Q_1 + Q_2 \xrightarrow{\alpha} Q'}$$

$$\textsc{Read}_1\ \frac{\{\alpha, \underline{\alpha}\} \cap \{\mu_1, \ldots, \mu_n\} \neq \emptyset}{\{\mu_1, \ldots, \mu_n\} \triangleright Q \xrightarrow{\alpha} \{\mu_1, \ldots, \mu_n\} \triangleright Q} \qquad \textsc{Read}_2\ \frac{Q \xrightarrow{\alpha} Q'}{\{\mu_1, \ldots, \mu_n\} \triangleright Q \xrightarrow{\alpha} Q'}$$

$$\textsc{Par}_1\ \frac{\alpha \notin A,\ Q_1 \xrightarrow{\alpha} Q_1'}{Q_1 \|_A Q_2 \xrightarrow{\alpha} \mathsf{clean}(Q_1' \|_A Q_2)} \qquad \textsc{Par}_2\ \frac{\alpha \in A,\ Q_1 \xrightarrow{\alpha} Q_1',\ Q_2 \xrightarrow{\alpha} Q_2'}{Q_1 \|_A Q_2 \xrightarrow{\alpha} \mathsf{clean}(Q_1' \|_A Q_2')}$$

$$\textsc{Rel}\ \frac{Q \xrightarrow{\alpha} Q'}{Q[\Phi] \xrightarrow{\Phi(\alpha)} Q'[\Phi]} \qquad \textsc{Rec}\ \frac{Q\{\mathsf{rec}\ x.\mathsf{unmark}(Q)/x\} \xrightarrow{\alpha} Q'}{\mathsf{rec}\ x.Q \xrightarrow{\alpha} Q'}$$

Table 1. Functional behaviour of PAFAS$_s$ processes

An essential idea of reading is that it does not change the state of a process and therefore does not block other actions. With the above operational semantics, we have $\{a\} \triangleright \{b\} \triangleright Q \xrightarrow{b} \{b\} \triangleright Q$ as well as $\{a\} \triangleright Q' + \{b\} \triangleright Q \xrightarrow{b} \{b\} \triangleright Q$, violating this idea; therefore, we exclude such processes.

Definition 5. (*activated basic actions*) $\mathcal{A}(Q) = \{\alpha \mid Q \xrightarrow{\alpha}\}$ is the set of *activated* (or enabled) actions of $Q \in \tilde{\mathbb{P}}$. $\mathcal{A}(Q, A) = \mathcal{A}(Q) \backslash A$ is the set of activated actions of Q when the environment prevents the actions in $A \subseteq \mathbb{A}_\tau$.

2.2 The temporal behaviour of PAFAS$_s$ processes

Our definition of timed behaviour is based on what we call (timed) refusal traces. Such a trace records, along a computation, which actions process Q performs ($Q \xrightarrow{\alpha} Q'$, $\alpha \in \mathbb{A}_\tau$) and which actions Q can refuse to perform when time

$$\textsc{Nil}_t\ \frac{}{\mathsf{nil} \xrightarrow{X}_r \mathsf{nil}} \qquad \textsc{Pref}_{t1}\ \frac{}{\alpha.P \xrightarrow{X}_r \underline{\alpha}.P} \qquad \textsc{Pref}_{t2}\ \frac{\alpha \notin X \cup \{\tau\}}{\underline{\alpha}.P \xrightarrow{X}_r \underline{\alpha}.P}$$

$$\textsc{Read}_t\ \frac{\mathcal{U}(\{\mu_1,\ldots,\mu_n\}) \cap (X \cup \{\tau\}) = \emptyset, Q \xrightarrow{X}_r Q'}{\{\mu_1,\ldots,\mu_n\} \triangleright Q \xrightarrow{X}_r \{\underline{\mu_1},\ldots,\underline{\mu_n}\} \triangleright Q'} \qquad \textsc{Sum}_t\ \frac{Q_i \xrightarrow{X}_r Q_i' \text{ for } i = 1,2}{Q_1 + Q_2 \xrightarrow{X}_r Q_1' + Q_2'}$$

$$\textsc{Rel}_t\ \frac{Q \xrightarrow{\Phi^{-1}(X \cup \{\tau\}) \setminus \{\tau\}}_r Q'}{Q[\Phi] \xrightarrow{X}_r Q'[\Phi]} \qquad \textsc{Rec}_t\ \frac{Q \xrightarrow{X}_r Q'}{\mathsf{rec}\ x.Q \xrightarrow{X}_r \mathsf{rec}\ x.Q'}$$

$$\textsc{Par}_t\ \frac{Q_i \xrightarrow{X_i}_r Q_i' \text{ for } i = 1,2, X \subseteq (A \cap (X_1 \cup X_2)) \cup ((X_1 \cap X_2) \setminus A)}{Q_1 \|_A Q_2 \xrightarrow{X}_r \mathsf{clean}(Q_1' \|_A Q_2')}$$

Table 2. Refusal transitional semantics of PAFAS$_s$ processes

elapses ($Q \xrightarrow{X}_r Q'$, $X \subseteq \mathbb{A}$). A transition like $Q \xrightarrow{X}_r Q'$ is called a (partial) *time-step*. The actions listed in X are not urgent; hence Q is justified in not performing them, but performing a time step instead. If $X = \mathbb{A}$ then Q is fully justified in performing this time-step; i.e., Q can perform it independently of the environment. In such a case, we say that Q performs a *full time-step* and write $Q \xrightarrow{1} Q'$; moreover, we often write $\underline{Q}$ for Q'. Our real interest is in runs where all time steps are full.

Definition 6. (*refusal transitional semantics*) The relations $\xrightarrow{X}_r \subseteq \tilde{\mathbb{P}} \times \tilde{\mathbb{P}}$ with $X \subseteq \mathbb{A}$ are defined by the inference rules in Table 2.

Rule $\textsc{Pref}_{t_1}$ says that a process $\alpha.P$ can let time pass and refuse to perform any action while rule $\textsc{Pref}_{t_2}$ says that a process $\underline{\alpha}.P$ can let time pass but cannot refuse the action α. Process $\underline{\tau}.P$ cannot let time pass and cannot refuse any action; in any context, $\underline{\tau}.P$ has to perform τ before time can pass further. Rule $\textsc{Par}_t$ defines which actions a parallel composition can refuse during a time-step. The intuition is that $Q_1 \|_A Q_2$ can refuse an action α if either $\alpha \notin A$ (Q_1 and Q_2 can do α independently) and both Q_1 and Q_2 can refuse α, or $\alpha \in A$ (Q_1 and Q_2 are forced to synchronise on α) and at least one of Q_1 and Q_2 can refuse α, i.e. can delay it. Thus, an action in a parallel composition is urgent (cannot be further delayed) only when all synchronising 'local' actions are urgent (also in this case we unmark the inactive urgencies). The other rules are as expected.

Example 1. Consider $P = (R \|_\emptyset W) \|_{\{r,w\}} V$, where $V = \mathsf{rec}\ x.(r.x + w.x)$, $R = \mathsf{rec}\ x.r.x$ and $W = \mathsf{rec}\ x.w.x$ model a variable (with values abstracted away), and the activities of repeatedly reading and writing such a variable, resp. By our operational rules, $V \xrightarrow{1} \underline{V} = \mathsf{rec}\ x.\ (\underline{r}.x + \underline{w}.x) \xrightarrow{r} V$ (each occurrence of x is replaced by V before the second transition by using unmark); hence, x is replaced by the original V as one would expect. Furthermore: $P \xrightarrow{1} ((\mathsf{rec}\ x.\ \underline{r}.x) \|_\emptyset (\mathsf{rec}\ x.\ \underline{w}.x)) \|_{\{r,w\}} \mathsf{rec}\ x.\ (\underline{r}.x + \underline{w}.x) \xrightarrow{r} P$. Here, the second

transition models the execution of action r by synchronising $\underline{R}$ and $\underline{V}$. These processes evolve into R and V, resp. and, as a side effect, the urgent w in $\underline{W}$ loses its urgency (due to function clean), since its synchronisation partner V offers a new, non-urgent synchronisation. The above behaviour can be repeated, demonstrating that readings can repeatedly delay and thus block w indefinitely.

Alternatively, we can model the action r as non-blocking with $V' = \mathsf{rec}x.\{r\}\triangleright (w.x)$. By our operational rules, $V' \xrightarrow{1} \underline{V'} = \mathsf{rec}\ x.(\{\underline{r}\} \triangleright (\underline{w}.x)) \xrightarrow{r} \mathsf{rec}\ x.(\{\underline{r}\} \triangleright (\underline{w}.x)) \xrightarrow{w} V'$. Hence: $P' \xrightarrow{1} Q = (\underline{R} \| _\emptyset \underline{W}) \|_{\{r,w\}} \underline{V'} \xrightarrow{r} Q' = (R \| _\emptyset \mathsf{rec}x.\underline{w}.x) \|_{\{r,w\}} \mathsf{rec}\ x.(\{r\} \triangleright (\underline{w}.x)) \xrightarrow{r} Q' \xrightarrow{w} (R \|_\emptyset W) \|_{\{r,w\}} V' = P'$. After the first occurrence of r (corresponding to a synchronisation between $\underline{R}$ and $\underline{V'}$), $\underline{R}$ becomes R and offers a new, non-urgent, instance of r to its partner; this causes the unmarking of the urgent r in $\underline{V'}$. Once in Q', we can either perform an r-action, evolving again into Q', or perform an action w and come back to P'. But we cannot perform 1, i.e. w is not delayed by r in contrast to P above. Moreover, from Q we can also perform w evolving directly to P'. In this step, the urgent r in $\underline{R}$ loses its urgency as above due to a new, non-urgent, synchronisation offered by V'. Therefore, repeated writes can delay the action r arbitrarily long, i.e. writing can still block reading.

3 Fairness and timing

In [5] we have defined fair traces in an intuitive, but very complex fashion in the spirit of [8] such that an action has to occur in an untimed run if it is enabled in every process of the run from some stage onward; then, we have characterised fair traces with transition sequences having infinitely many full time steps, so called *non-Zeno timed execution sequences*, generalising [4]. Here, due to lack of space, we omit the definition; the following characterisation will serve as definition of fair traces for the remainder of this paper.

Theorem 1. (*fair traces*) Let $P_0 \in \mathbb{P}_1$ be read-proper and $\alpha_0, \alpha_1, \alpha_2, \ldots \in \mathbb{A}_\tau$. A trace of P_0 (i.e. a sequence of actions) is *fair (w.r.t. actions)* if it is the sequence of actions in a non-Zeno timed execution sequence. In detail:

1. A finite trace $\alpha_0\alpha_1 \ldots \alpha_n$ is fair if and only if there exists a timed execution sequence $P_{i_0} \xrightarrow{1} Q_{i_0} \xrightarrow{v_0} P_{i_1} \xrightarrow{1} Q_{i_1} \xrightarrow{v_1} P_{i_2} \ldots P_{i_{m-1}} \xrightarrow{1} Q_{i_{m-1}} \xrightarrow{v_{m-1}} P_{i_m} \xrightarrow{1} Q_{i_m} \xrightarrow{1} Q_{i_m} \ldots$, where $P_{i_0} = P_0$ and $v_0\, v_1 \ldots v_{m-1} = \alpha_0\, \alpha_1 \ldots \alpha_n$;
2. an infinite trace $\alpha_0\alpha_1\alpha_2 \ldots$ is fair if and only if there exists a timed execution sequence $P_{i_0} \xrightarrow{1} Q_{i_0} \xrightarrow{v_0} P_{i_1} \xrightarrow{1} Q_{i_1} \xrightarrow{v_1} P_{i_2} \cdots P_{i_m} \xrightarrow{1} Q_{i_m} \xrightarrow{v_m} P_{i_{m+1}} \cdots$, where $P_{i_0} = P_0$ and $v_0\, v_1 \ldots\, v_m \ldots = \alpha_0\, \alpha_1 \ldots\, \alpha_i \ldots$.

Example 2. Consider again $P = (R \|_\emptyset W) \|_{\{r,w\}} V$ from Example 1 and the trace consisting of infinitely many r's. This is fair, also when considering w because, at each transition, process V offers a "fresh" action w for synchronisation – each time an action r is performed, a new instance of w is produced. We can use timing to see this formally. As we noted in Example 1:

$P \xrightarrow{1} ((\mathsf{rec}\ x.\ \underline{r}.x) \parallel_\emptyset (\mathsf{rec}\ x.\ \underline{w}.x)) \parallel_{\{r,w\}} \mathsf{rec}\ x.\ (\underline{r}.x + \underline{w}.x) \xrightarrow{r} P$. If we repeat this infinitely often, we get a non-Zeno timed execution sequence related to the trace of infinitely many r's. Thus, fairness of actions allows computations along which repeated reading of a variable indefinitely blocks another process trying to write to it (and vice versa for repeated writing).

This is the reason why some fair runs of Dekker's algorithm violate liveness (see below) when using standard PAFAS [3]. This problem is not specific to our setting or to our notion of fairness. In [13], Raynal writes about this algorithm that possibly, " if P_i is a very fast repetitive process which ... keeps entering its critical section, ... P_j cannot set $flag[j]$ to true, being prevented from doing so by P_i's reading of the variable." He observes that liveness of the algorithm therefore depends on the liveness of the hardware. This is exactly the sort of consideration for which we have a formal treatment: read prefixes say that the hardware guarantees that at least infinite reading cannot block writing.

We can prevent this kind of unwanted behaviour by modelling the action r as reading (see Example 1). Indeed, a run from $P' = (R \parallel_\emptyset W) \parallel_{\{r,w\}} V'$ consisting of infinitely many r's is not fair, since we can have at most one time step along such a run: e.g. $P' \xrightarrow{1} (\underline{R} \parallel_\emptyset \underline{W}) \parallel_{\{r,w\}} \underline{V'} = Q \xrightarrow{r} Q' \xrightarrow{r} \ldots \xrightarrow{r} Q' \ldots$, where Q' does not allow a full time step. Now, fairness of actions ensures that a process trying to write the variable will eventually do so. On the contrary, a run from P' consisting of infinitely many w only is still fair (by Example 1, $P' \xrightarrow{1} Q \xrightarrow{w} P'$) and, hence, repeated writing of a variable can block another process trying to read it. This kind of behaviour can be prevented if the variable is modelled as $V'' = \{r, w\} \triangleright \mathsf{nil}$, which only makes sense if the same value is written, not changing the system state.

Remark 1. To see that re-writing the same value can indeed be non-blocking in practice, consider the *two-phase locking protocol* implementing a lock system where each transaction may only access a data item if it holds a lock on that item. There are two possible modes of locks: *shared* and *exclusive*. If a transaction T holds a *shared mode* lock (an S-lock) on data item q, then T may read – but not write – q. On the other hand, a transaction with an *exclusive mode* lock (an X-lock) on q can both read and write it. Multiple S-locks are allowed on a single data item, but only one X-lock can be acquired for it. This allows multiple reads (which do not create serialisability conflicts) as in our modelling of variables, but writing prevents reading or another writing (which would create conflicts).

A transaction acquires new locks only during the so-called *growing phase*. All the locks acquired in the growing phase can be released only during a subsequent phase, called *shrinking phase*. Furthermore, an S-lock can be upgraded to X during the growing phase and, similarly, an X-lock can be downgraded to S during the shrinking phase. The idea here is that – during the growing phase – a transaction, instead of holding an X-lock on an item that it does not need to write yet, can hold an S-lock until the point where modifications to the old value begin, in order to allow other transactions to read the old value for longer. This can be used for a "reading first" implementation of writing: each write operation first reads the old value (this read requires an S-lock for the variable and can be

done concurrently with other read operations) and then only writes a value only if it is really a new one (in this latter case, the S-lock has to be upgraded to an X-lock). This way, a re-writing of the same value is indeed non-blocking.

4 Dekker's algorithm and its liveness property

In this section we briefly describe Dekker's MUTEX algorithm. There are two processes $\mathtt{P}_1$ and $\mathtt{P}_2$, two Boolean-valued variables b_1 and b_2, whose initial values are *false*, and a variable k, which may take the values 1 and 2 and whose initial value is arbitrary. Informally, the b variables are "request" variables and k is a "turn" variable: b_i is *true* if $\mathtt{P}_i$ is requesting entry to its critical section and k is i if it is $\mathtt{P}_i$'s turn to enter its critical section. Only $\mathtt{P}_i$ writes b_i, but both processes read it. The ith process (with $i = 1, 2$) can be described as follows, where j is the index of the other process:

```
while true do begin
    ⟨noncritical section⟩;
    b_i = true;
    while b_j do if k = j then begin
          b_i := false;   while k = j do skip;   b_i := true;
    end;
    ⟨critical section⟩;
    k := j;   b_i := false;
end;
```

4.1 Translating the Algorithm into PAFAS$_s$ Processes

In our translation of the algorithm into PAFAS$_s$, we use essentially the same coding as Walker in [16]. Each program variable is represented as a family of processes. For instance, the process $\mathsf{B}_1(\mathit{false})$ denotes the variable b_1 with value *false*. The *sort* of the process $\mathsf{B}_1(\mathit{false})$ (i.e. the set of actions it can ever perform) is the set $\{b_1 rf, b_1 rt, b_1 wf, b_1 wt\}$ where $b_1 rf$ and $b_1 rt$ represent the actions of reading the values *false* and *true* from b_1, $b_1 wf$ and $b_1 wt$ represent, resp., the writing of the values *false* and *true* into b_1. Let $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$ and $\mathbb{K} = \{1, 2\}$.

Definition 7. (*the algorithm*) Let $i \in \{1, 2\}$. We define the processes representing program variables as follows:

$\mathsf{B}_i(\mathit{false}) = \{b_i rf, b_i wf\} \triangleright b_i wt.\mathsf{B}_i(\mathit{true})$ $\qquad$ $\mathsf{K}(1) = \{\mathit{kr1}, \mathit{kw1}\} \triangleright \mathit{kw2}.\mathsf{K}(2)$

$\mathsf{B}_i(\mathit{true}) = \{b_i rt, b_i wt\} \triangleright b_i wf.\mathsf{B}_i(\mathit{false})$ $\qquad$ $\mathsf{K}(2) = \{\mathit{kr2}, \mathit{kw2}\} \triangleright \mathit{kw1}.\mathsf{K}(1)$

Let $B = \{b_i rf, b_i rt, b_i wf, b_i wt \mid i \in \{1, 2\}\} \cup \{\mathit{kr1}, \mathit{kr2}, \mathit{kw1}, \mathit{kw2}\}$ be the union of the sorts of all variables and Φ_B the relabelling function such that $\Phi_B(\alpha) = \tau$ if $\alpha \in B$ and $\Phi_B(\alpha) = \alpha$ if $\alpha \notin B$. Given $b_1, b_2 \in \mathbb{B}$, $k \in \mathbb{K}$, we define $\mathsf{PV}(b_1, b_2, k) = (\mathsf{B}_1(b_1) \parallel_\emptyset \mathsf{B}_2(b_2)) \parallel_\emptyset \mathsf{K}(k)$. Processes $\mathtt{P}_1$ and $\mathtt{P}_2$ are represented by the following PAFAS$_s$ processes where the actions $\mathtt{req}_i$ and $\mathtt{cs}_i$ indicate the request to enter and the execution of the critical section by the process $\mathtt{P}_i$.

$$P_1 = \mathtt{req}_1.b_1\,wt.P_{11} + \tau.P_1 \qquad P_2 = \mathtt{req}_2.b_2\,wt.P_{21} + \tau.P_2$$
$$P_{11} = b_2\,rf.P_{14} + b_2\,rt.P_{12} \qquad P_{21} = b_1\,rf.P_{24} + b_1\,rt.P_{22}$$
$$P_{12} = kr1.P_{11} + kr2.b_1\,wf.P_{13} \qquad P_{22} = kr2.P_{21} + kr1.b_2\,wf.P_{23}$$
$$P_{13} = kr1.b_1\,wt.P_{11} + kr2.P_{13} \qquad P_{23} = kr2.b_2\,wt.P_{21} + kr1.P_{23}$$
$$P_{14} = \mathtt{cs}_1.kw2.b_1\,wf.P_1 \qquad P_{24} = \mathtt{cs}_2.kw1.b_2\,wf.P_2$$

Now, we define the algorithm as $Dekker = ((P_1 \parallel P_2) \parallel_B \mathsf{PV}(false, false, 1))[\Phi_B]$. The sort of *Dekker* is the set $\mathbb{A}_d = \{\mathtt{req}_i, \mathtt{cs}_i \mid i = 1, 2\}$.

A MUTEX algorithm like Dekker's satisfies *liveness* if, in every fair trace, each $\mathtt{req}_i$ is followed by the respective $\mathtt{cs}_i$. Since no process should be forced to request by the fairness assumption, P_i has the alternative of an internal move, i.e. staying in its noncritical section.

4.2 Liveness violations and catastrophic cycles

Based on PAFAS, a testing-based faster-than relation has been defined in [7] that compares processes according to their worst-case efficiency. In [6], this testing-approach is adapted to a setting where user behaviour is known to belong to a very specific, but often occurring class of request-response behaviours: processes serving these users receive requests via an action *in* and provide a response *out* for each *in*-action; it is shown how to determine an asymptotic performance measure for finite-state processes of this kind. This result only holds for request-response processes that pass certain sanity checks: they must not produce more responses than requests, and they must allow requests and provide responses in finite time. While the first requirement can easily be read off from the transition system, violation of the latter requirement is characterised as the existence of so-called *catastrophic cycles* in a reduced transition system denoted $\mathsf{rRTS}(P)$.

The *refusal transition system* of P consists of all transitions $Q \xrightarrow{\alpha} Q'$ and $Q \xrightarrow{X}_r Q'$, where Q is reachable from P via such transitions. $\mathsf{rRTS}(P)$ is obtained by removing all time steps except those $Q \xrightarrow{X}_r Q'$ where either $X = \{out\}$ and Q has some *pending out*-action or $X = \{in, out\}$; then, all processes not reachable any more are deleted as well. In the case $X = \{out\}$, some *in* has not received a response and the user is waiting for an *out*, but the process can still delay this, while being willing to accept another request immediately. The case $X = \{in, out\}$ corresponds to a full time step. A cycle is catastrophic if it contains a time step but no *in*- or *out*-transition, such that time can pass without end but without any useful actions; see [6] for more details.

A tool has been developed for automatically checking whether a process of (original) PAFAS has a catastrophic cycle with the algorithm described in [6], and only recently it has been adapted to a setting with reading actions. We will now give a result that allows us to decide whether *Dekker* is live using this tool.

The tool cannot be applied directly: first, *Dekker* has more than two actions; second, it can perform a full time step followed by the two internal actions of P_1 and P_2 giving a catastrophic cycle, which is not relevant for the liveness property. Consequently, we modify *Dekker* to obtain a new process $Dekker_{io}$ as follows:

we change the actions $\mathtt{req}_1$ and $\mathtt{cs}_1$ into τ actions, we delete the τ-summand of P_2 (see Definition 7) and, finally, we change the actions $\mathtt{req}_2$ and $\mathtt{cs}_2$ in *in* and *out*, respectively. With this, we get the following result and corollary:

Theorem 2. *Dekker* is live iff $Dekker_{io}$ does not have catastrophic cycles.

Corollary 1. *Dekker* is live.

To further stress the impact of introducing non-blocking actions in PAFAS (and in general in modelling concurrent systems), we can obtain $Dekker_\ell$ from *Dekker* by regarding writing the same value as blocking. We prove in [5] that this slight change has a decisive impact on liveness:

Theorem 3. $Dekker_\ell$ is not live.

References

1. P. Bouyer, S. Haddad, P.A. Reynier. Timed Petri Nets and Timed Automata: On the Discriminating Power of Zeno Sequences. Proc. of ICALP'06, LNCS 4052, pp. 420-431, 2006.
2. S. Christensen, N. D. Hansen. Coloured Petri nets extended with place capacities, test arcs, and inhibitor arcs. In Applications of Theory of Petri Nets, LNCS 691, pp. 186-205, 1993.
3. F. Corradini, M.R. Di Berardini, and W. Vogler. Checking a Mutex Algorithm in a Process Algebra with Fairness. Proc. of CONCUR '06, pp. 142-157, LNCS 4137, 2006.
4. F. Corradini, M.R. Di Berardini and W. Vogler. Fairness of Actions in System Computations. *Acta Informatica* **43**, pp. 73 130, 2006.
5. F. Corradini, M.R. Di Berardini and W. Vogler. Time and Fairness in a Process Algebra with Non-Blocking Reading. Technical Report 2008-13, Institute of Computer Science, University of Augsburg, 2008.
6. F. Corradini, W. Vogler. Measuring the Performance of Asynchronous Systems with PAFAS. *Theoretical Computer Science*, **335**, pp. 187-213, 2005.
7. F. Corradini, W. Vogler, and L. Jenner. Comparing the Worst-Case Efficiency of Asynchronous Systems with PAFAS. *Acta Informatica* **38**, pp. 735-792, 2002.
8. G. Costa, C. Stirling. Weak and Strong Fairness in CCS. *Information and Computation* **73**, pp. 207-244, 1987.
9. F. Crazzolara, G. Winskel. Events in security protocols. In Proc. of 8th ACM conference on Computer and Communication Security, CCS'01, pp. 96-105, 2001.
10. R. Milner. *Communication and Concurrency*. International series in computer science, Prentice Hall International, 1989.
11. U. Montanari, F. Rossi. Contextual net. *Acta Informatica* **32**, pp. 545-596, 1995.
12. U. Montanari, F. Rossi. Contextual occurrence nets and concurrent constraints programming. In Proc. of Graph Transformation in Computer Science, LNCS 776, pp. 280-295, 1994
13. M. Raynal. *Algorithms for Mutual Exclusion*. North Oxford Academic, 1986.
14. G. Ristori. Modelling Systems with Shared Resources via Petri Nets. PhD thesis, Department of Computer Science, University of Pisa, 1994.
15. W. Vogler. Efficiency of Asynchronous Systems, Read Arcs and the MUTEX-problem. *Theoretical Computer Science* 275(1-2), pp. 589-631, 2002.
16. D.J. Walker. Automated Analysis of Mutual Exclusion algorithms using CCS. *Formal Aspects of Computing* **1**, pp. 273-292, 1989.