Checking a Mutex Algorithm in a Process Algebra with Fairness

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. In earlier work, we have shown that two variants of weak fairness can be expressed comparatively easily in the timed process algebra PAFAS. To demonstrate the usefulness of these results, we complement work by Walker [11] and study the liveness property of Dekker's mutual exclusion algorithm within our process algebraic setting. We also present some results that allow to reduce the state space of the PAFAS process representing Dekker's algorithm, and give some insight into the representation of fair behaviour in PAFAS.

1 Introduction

This paper was inspired by the work of Walker [11] who aimed at automatically verifying six mutual exclusion algorithms – including Dekker's. Walker translated the algorithms into the process algebra CCS [9] and then verified with the Concurrency Workbench [1] that all of them satisfy the *safety property* that the two competing processes are never in their critical sections at the same time.

The *liveness property* that a requesting process will always eventually enter the critical section is more difficult to verify, since one has to assume some fairness, which is not so easy to do in a process algebraic setting; with respect to the verification of liveness, Walker was less successful.

Costa and Stirling [6,7] have studied some notions of fairness in a process algebra. While their formalisation captures the intuition of fairness faithfully, it is technically involved and leads to processes with infinite state spaces – at least for processes that have an infinite computation. In [2,3], we have defined fair runs in the spirit of Costa and Stirling and characterised them in the timed process algebra PAFAS [5] as those runs that take infinitely long; here, processes that are finite state in a standard process algebra without time still have a finite transition system in the setting where fairness can be studied. The present paper complements the work by Walker, taking the liveness of Dekker's algorithm as a case study to demonstrate how our approach to fairness can be used.

Attempting the verification of the liveness property, Walker used the following version in [11] – which could be expressed as a modal mu-calculus formula and checked with the Concurrency Workbench:

C. Baier and H. Hermanns (Eds.): CONCUR 2006, LNCS 4137, pp. 142–157, 2006.

Whenever at some point in a run the process $\mathtt{P}_i$ *requests the execution of its critical section, then in any continuation of that run from that point in which between them the processes execute an infinite number of critical sections,* $\mathtt{P}_i$ *performs its critical section at least once.*

The fairness (or progress) assumed here is that infinitely often a critical section is entered. This assumption allows a run where one process enters its critical section repeatedly, while the other requests the execution of its critical section, but then – for no good reason at all – refuses to take the necessary steps to actually enter it. Thus, it is maybe not so surprising that four of the six mutual exclusion algorithms – including Dekker's – fail to satisfy this property. Walker then discusses how fairness could be assumed to enable a proof of liveness, but the ideas discussed could not be expressed for use of the Concurrency Workbench.

Here, we will model Dekker's algorithm in the CCS-type process algebra PAFAS and study whether all fair runs satisfy the liveness property. Actually, we consider two versions of PAFAS. The first one is suitable for (weak) *fairness of actions*, i.e. in a fair run each enabled action must be performed or disabled eventually; if this action is a synchronisation, then the action is already disabled if one partner of this synchronisation offers a different instance of the action. As a consequence, repeated accesses to a variable can block another access, and for this reason some fair runs of Dekker's algorithm violate liveness; this is not so different from Walker's result, but we can point to a realistic reason for the failure, namely the blocking of a variable. We provide two fair runs, one in which one process repeatedly enters its critical section while the other is stuck, and one where both processes are stuck.

It is equally realistic to assume that access to a variable cannot be blocked indefinitely. In the second version of PAFAS, we deal with (weak) *fairness of components*, i.e. in a fair run each enabled component must be performed or disabled eventually. Thus, if a process wants to read a binary variable, it will offer two read-actions (one for each value); if none of these is performed, then in every future state one or the other will be enabled, i.e. the process will be enabled indefinitely; fairness now implies that the process actually will read the variable eventually. Assuming fairness of components, we will show that Dekker's algorithm indeed satisfies the liveness property. In this proof, we have to take into account all possible derivatives reachable from *Dekker* along fair computations. In particular, we will consider those states where one process has just performed a request to enter a critical section, and show that from those states the respective process does eventually enter the critical section.

Modelling fairness involves a certain blow-up of the state space, so for a proof by hand the number of states we had to deal with was rather large. Consequently, to manage the proof, we had to rely on structural properties of the processes, which may be of interest independently of the main aims of this paper. Previously, we have characterised fair runs as those action sequences that arise from timed computations with infinitely many unit time steps by deleting these time steps. Our first result states that we can restrict attention to a particular subclass of such timed computations and still cover all fair runs. A considerable reduction

of states comes from switching some components to "permanently lazy", i.e. to require fairness only for the other components. In our case study, the "permanently lazy components" correspond to the variables; so this is a very realistic change, since it seems natural that only the processes are active, while a variable never forces to be read or to be written. In general, switching some components to permanently lazy gives an overapproximation for the fair runs, and it is clearly sufficient to prove a desired property for this possibly larger set of runs. Finally, we take advantage of symmetries in the *Dekker* algorithm. The two processes that compete for the execution of their critical section, indeed, have a symmetric structure so that their derivatives follow a symmetric pattern. Thus, we check liveness of a generic fair-reachable derivative to deduce the same property of the symmetric one; see [4] for many details omitted here. These observations have allowed a proof by hand. We believe, however, that they are not specific to this work but really add some general knowledge to the theory of PAFAS useful to be embedded within an automatic tool for the verification based on fairness.

2 Fairness and PAFAS

We now recall PAFAS, its timed behaviour and the fairness notions we consider, namely fairness of actions and components. Instead of using the very involved direct formalisations of fairness in the spirit of [6,7], we define the two types of fair traces on the basis of our characterisations with everlasting timed execution sequences in the two respective versions of PAFAS.

2.1 Fairness of Actions and PAFAS

We use the following notation: $\mathbb{A}$ is an infinite set of basic actions. An additional action τ is used to represent internal activity, which is unobservable for other components. We define $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau\}$. Elements of $\mathbb{A}$ are denoted by $a, b, c, \ldots$ and those of $\mathbb{A}_\tau$ are denoted 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. $\mathcal{X}$ is the set of process variables, used for recursive definitions. Elements of $\mathcal{X}$ are denoted by $x, y, z, \ldots$ $\Phi : \mathbb{A}_\tau \to \mathbb{A}_\tau$ 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 $\alpha \notin A$.

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

$$P ::= \mathsf{nil} \mid x \mid \alpha.P \mid P + P \mid P \|_A P \mid P[\Phi] \mid \mathsf{rec}\, x.P$$

where nil is a constant, $x \in \mathcal{X}$, $\alpha \in \mathbb{A}_\tau$, Φ is a general relabelling function and $A \subseteq \mathbb{A}$ possibly infinite. We assume that recursion is guarded (see below).

The set $\tilde{\mathbb{P}}$ of (general) *(timed) process terms* is generated by the following grammar:

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

where $P \in \tilde{\mathbb{P}}_1$, $x \in \mathcal{X}$, $\alpha \in \mathbb{A}_\tau$, Φ is a general relabelling function and $A \subseteq \mathbb{A}$ possibly infinite. We assume that the recursion is *guarded*, i.e. for $\mathsf{rec}x.Q$ variable x 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 *guarded* if each occurrence of a variable is guarded in this sense. A timed process term Q is closed, if every variable x in Q is bound by the corresponding $\mathsf{rec}x$-operator; such Q in $\tilde{\mathbb{P}}$ and $\tilde{\mathbb{P}}_1$ are simply called *processes* and *initial processes* resp., and their sets are denoted by $\mathbb{P}$ and $\mathbb{P}_1$ resp.[1]

Initial processes are just standard processes of a standard process algebra. General processes are defined here such that they include all processes reachable from the initial ones according to the operational semantics to be defined below.

We can now define the set of activated actions in a process term. Given a process term Q, $\mathcal{A}(Q, A)$ will denote the set of the *activated* (or enabled) actions of Q when the environment prevents the actions in A.

Definition 2. (*activated basic actions*) Let $Q \in \tilde{\mathbb{P}}$ and $A \subseteq \mathbb{A}$. The set $\mathcal{A}(Q, A)$ is defined by induction on Q.

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

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

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

Par: $\mathcal{A}(Q_1 \parallel_B Q_2, A) = \mathcal{A}(Q_1, A \cup A') \cup \mathcal{A}(Q_2, A \cup A'')$
where $A' = (\mathcal{A}(Q_1) \backslash \mathcal{A}(Q_2)) \cap B$ and $A'' = (\mathcal{A}(Q_2) \backslash \mathcal{A}(Q_1)) \cap B$

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

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

The *activated actions* of Q are defined as $\mathcal{A}(Q, \emptyset)$ which we abbreviate to $\mathcal{A}(Q)$.

Definition 3. (*urgent activated action*) Let $Q \in \tilde{\mathbb{P}}$ and $A \subseteq \mathbb{A}$. The set $\mathcal{U}(Q, A)$ is defined as in Definition 2 when $\mathcal{A}(\_)$ is replaced by $\mathcal{U}(\_)$ and the Pref-rule is replaced by the following one:

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

The *urgent activated actions* of Q are defined as $\mathcal{U}(Q) = \mathcal{U}(Q, \emptyset)$

The operational semantics exploits two functions on process terms: $\mathsf{clean}(\_)$ and $\mathsf{unmark}(\_)$. Function $\mathsf{clean}(\_)$ removes *all inactive urgencies* in a process term

[1] In [5], we prove that $\mathbb{P}_1$ processes do not have time-stops; i.e. every finite process run can be extended such that time grows unboundedly. This result was proven for a different operational semantics than that defined in this paper but a similar proof applies also in the current setting.

$Q \in \tilde{\mathbb{P}}$. When a process evolves and a synchronized action is no longer urgent or enabled in some synchronization partner, then it should also lose its urgency in the others; the corresponding change of markings is performed by clean, where again set A in $\mathsf{clean}(Q, A)$ denotes the set of actions that are not enabled or urgent due to restrictions of the environment. Function $\mathsf{unmark}(\_)$ simply removes all urgencies (inactive or not) in a process term $Q \in \tilde{\mathbb{P}}$. We provide the formal definition of the former function. The second one is as expected.

Definition 4. (*cleaning inactive urgencies*) Given a process term $Q \in \tilde{\mathbb{P}}$ we define $\mathsf{clean}(Q)$ as $\mathsf{clean}(Q, \emptyset)$ where, for a set $A \subseteq \mathbb{A}$, $\mathsf{clean}(Q, A)$ is defined as:

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

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

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

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

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

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

The Functional Behaviour of PAFAS Process. The transitional semantics describing the functional behaviour of PAFAS processes indicates which basic actions they can perform.

Definition 5. (*Functional operational semantics*) The following SOS-rules define the action transition relations $\xrightarrow{\alpha} \subseteq (\tilde{\mathbb{P}} \times \tilde{\mathbb{P}})$ for $\alpha \in \mathbb{A}_\tau$. 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 $(Q, Q') \in \xrightarrow{\alpha}$, and similar conventions will apply later on.

$$\textsc{Pref}_{a1}\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \textsc{Pref}_{a2}\frac{}{\underline{\alpha}.P \xrightarrow{\alpha} P} \qquad \textsc{Sum}_a\frac{Q_1 \xrightarrow{\alpha} Q'}{Q_1 + Q_2 \xrightarrow{\alpha} Q'}$$

$$\textsc{Par}_{a1}\frac{\alpha \notin A,\ Q_1 \xrightarrow{\alpha} Q_1'}{Q_1 \|_A Q_2 \xrightarrow{\alpha} \mathsf{clean}(Q_1' \|_A Q_2)} \qquad \textsc{Par}_{a2}\frac{\alpha \in A,\ Q_1 \xrightarrow{\alpha} Q_1',\ Q_2 \xrightarrow{\alpha} Q_2'}{Q_1 \|_A Q_2 \xrightarrow{\alpha} \mathsf{clean}(Q_1' \|_A Q_2')}$$

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

Additionally, there are symmetric rules for $\textsc{Par}_{a1}$ and $\textsc{Sum}_a$ for actions of Q_2. For an *initial* process P_0, we say that a finite or infinite sequence $\alpha_0\alpha_1 \ldots$ of actions from $\mathbb{A}_\tau$ is a *trace* of P_0, if there is a sequence $P_0 \xrightarrow{\alpha_0} P_1 \xrightarrow{\alpha_1} \ldots$ of action transitions, possibly ending with a process P_n.

The Temporal Behaviour of PAFAS Process. Now, we consider transitions corresponding to the passage of one unit of time. The function urgent marks all *enabled* actions of a process as urgent when a time step is performed. Before the next time step, all such actions must occur or get disabled.

Definition 6. (*time step, timed execution sequences*) For $P \in \tilde{\mathbb{P}}_1$, we write $P \xrightarrow{1} Q$ when $Q = \mathsf{urgent}(P)$, where $\mathsf{urgent}(P)$ abbreviates $\mathsf{urgent}(P, \emptyset)$ and $\mathsf{urgent}(P, A)$ is defined as:

Nil, Var:	$\mathsf{urgent}(\mathsf{nil}, A) = \mathsf{nil}, \qquad \mathsf{urgent}(x, A) = x$
Pref:	$\mathsf{urgent}(\alpha.P, A) = \begin{cases} \underline{\alpha}.P & \text{if } \alpha \notin A \\ \alpha.P & \text{otherwise} \end{cases}$
Sum:	$\mathsf{urgent}(P_1 + P_2, A) = \mathsf{urgent}(P_1, A) + \mathsf{urgent}(P_2, A)$
Par:	$\mathsf{urgent}(P_1 \parallel_B P_2, A) = \mathsf{urgent}(P_1, A \cup A') \parallel_B \mathsf{urgent}(P_2, A \cup A'')$ where $A' = (\mathcal{A}(P_1) \backslash \mathcal{A}(P_2)) \cap B$ and $A'' = (\mathcal{A}(P_2) \backslash \mathcal{A}(P_1)) \cap B$
Rel:	$\mathsf{urgent}(P[\Phi, A) = \mathsf{urgent}(P, \Phi^{-1}(A))[\Phi]$
Rec:	$\mathsf{urgent}(\mathsf{rec}\, x.P, A) = \mathsf{rec}\, x.\, \mathsf{urgent}(P, A)$

For an initial process P_0, we say that a sequence of transitions $\gamma = P_0 \xrightarrow{1} Q_0 \xrightarrow{\lambda_1} \ldots$ with $\lambda_i \in \mathbb{A}_\tau \cup \{1\}$ is a *timed execution sequence* if it is an infinite sequence of action transitions and time steps (starting with a time step)[2]. A timed execution sequence is *everlasting* in the sense of having infinitely many time steps if and only if it is *non-Zeno*; a Zeno run would have infinitely many actions in a finite amount of time.

Fairness of Actions and Timing. We can now define the (weakly) fair traces in terms of non-Zeno execution sequences.

Definition 7. (*fair traces*) Let $P_0 \in \mathbb{P}_1$ and $\alpha_0, \alpha_1, \alpha_2, \ldots \in \mathbb{A}_\tau$. A trace of P_0 is *fair (w.r.t. fairness of actions)* if it can be obtained as 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 $v_0\, v_1 \ldots\, v_m\, \ldots = \alpha_0\, \alpha_1\, \ldots\, \alpha_i \ldots$.

This is a characterisation for fair traces obtained in [2] on the basis of a more intuitive, but very complex definition of fair traces in the spirit of [6,7].

2.2 Fairness of Components and PAFAS$^{\mathbf{c}}$

In this section, we concentrate on weak *fairness of components*. We have found a suitable variation of PAFAS and its semantics which allows us to characterize Costa and Stirling's fairness of components again in terms of a simple filtering

[2] Note that a maximal sequence of such transitions/steps is never finite, since for $\gamma = Q_0 \xrightarrow{\lambda_0} Q_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} Q_n$, we have $Q_n \xrightarrow{\alpha}$ or $Q_n \xrightarrow{1}$ (see Proposition 3.13 in [2]).

of system executions. Conceptually, we proceed analogously to Section 2.1, but a number of technical changes are needed. Since we associate time bounds to components in the present section, we may also mark - besides prefixes - the other dynamic operator + as urgent: a process $P + Q$ becomes $P \underline{+} Q$ after a time step. This variant of PAFAS is called PAFAS$^{\mathbf{c}}$ henceforth.

Definition 8. (*timed process terms*) Let $\tilde{\mathbb{P}}_1$ be the set of *initial timed process terms* as given in Definition 1. The set $\tilde{\mathbb{P}}_c$ of (component-oriented) *timed process terms* is generated by the grammar:

$$Q ::= P \mid \underline{\alpha}.P \mid P \underline{+} P \mid Q \|_A Q \mid Q[\Phi] \mid \mathsf{rec}\, x.Q$$

where $P \in \tilde{\mathbb{P}}_1$, $x \in \mathcal{X}$, $\alpha \in \mathbb{A}_\tau$, Φ is a general relabelling function, and $A \subseteq \mathbb{A}$ possibly infinite. Again, we assume that recursion is always guarded. The set of closed timed process terms in $\tilde{\mathbb{P}}_c$, simply called *processes* is denoted by $\mathbb{P}_c$.

Function $\mathcal{A}(\_)$ on process terms, returns the *activated* (or enabled) actions of a process term.

Definition 9. (*activated basic actions*) Let $Q \in \tilde{\mathbb{P}}_c$ and $A \subseteq \mathbb{A}$. The set $\mathcal{A}(Q, A)$ can be defined as in Definition 2 when rule Sum is replaced as follows:

Sum: $\quad \mathcal{A}(P_1 + P_2, A) = \mathcal{A}(P_1 \underline{+} P_2, A) = \mathcal{A}(P_1, A) \cup \mathcal{A}(P_2, A)$

The Operational Behaviour of PAFAS$^{\mathbf{c}}$ Processes. A new definition of function $\mathsf{clean}(\_)$ is needed

Definition 10. (*cleaning inactive urgencies*) For a process $Q \in \tilde{\mathbb{P}}_c$, define $\mathsf{clean}(Q)$ as $\mathsf{clean}(Q, \emptyset)$, $A \subseteq \mathbb{A}$, $\mathsf{clean}(Q, A)$ is defined as in Definition 4 where rules Sum and Par are replaced by:

Sum: $\mathsf{clean}(P_1 + P_2, A) = P_1 + P_2$

$$\mathsf{clean}(P_1 \underline{+} P_2, A) = \begin{cases} P_1 + P_2 & \text{if } \mathcal{A}(P_1) \cup \mathcal{A}(P_2) \subseteq A \\ P_1 \underline{+} P_2 & \text{otherwise} \end{cases}$$

Par: $\mathsf{clean}(Q_1 \parallel_B Q_2, A) = \mathsf{clean}(Q_1, A \cup A') \parallel_B \mathsf{clean}(Q_2, A \cup A'')$
where $\quad A' = (\mathcal{A}(Q_1) \backslash \mathcal{A}(Q_2)) \cap B$ and $A'' = (\mathcal{A}(Q_2) \backslash \mathcal{A}(Q_1)) \cap B$

Definition 11. (*Functional operational semantics*) The functional operational semantics for $\tilde{\mathbb{P}}_c$-terms is as in Definition 5 where $\rightarrow$ is replaced by $\longmapsto$ and rule Sum_a (and symmetrically its symmetric rules) are replaced by:

$$\textsc{Sum}_{a1} \frac{P_1 \stackrel{\alpha}{\longmapsto} P_1'}{P_1 + P_2 \stackrel{\alpha}{\longmapsto} P_1'} \qquad \textsc{Sum}_{a2} \frac{P_1 \stackrel{\alpha}{\longmapsto} P_1'}{P_1 \underline{+} P_2 \stackrel{\alpha}{\longmapsto} P_1'}$$

and function clean is the one in Definition 10. Consequently, traces out of an initial process P_0 consider $\longmapsto$ (instead of $\rightarrow$).

The Temporal Behaviour of PAFAS$^{\mathbf{C}}$ Process. As in Section 2.1, we define *timed execution sequences* to be infinite sequences of action transitions and time steps starting at some initial process P_0 (again a maximal sequence of such transitions/steps starting is never finite) and the property *non-Zeno*, where:

Definition 12. (*time step, timed execution sequence*) For $P \in \tilde{\mathbb{P}}_1$, we write $P \stackrel{1}{\longmapsto} Q$ when $Q = \mathsf{urgent}(P)$, where $\mathsf{urgent}(P)$ abbreviates $\mathsf{urgent}(P, \emptyset)$ and $\mathsf{urgent}(P, A)$ is defined as in Definition 6 but rule Sum is replaced as follows:

$$\text{Sum: } \mathsf{urgent}(P_1 + P_2, A) = \begin{cases} P_1 \underline{+} P_2 & \text{if } (\mathcal{A}(P_1) \cup \mathcal{A}(P_2)) \backslash A \neq \emptyset \\ P_1 + P_2 & \text{otherwise} \end{cases}$$

Fairness of Components and Timing. As in Section 2.1, we can now define (weak) *fairness w.r.t. components* in terms of non-Zeno timed execution sequences. In fact, *fair traces* (w.r.t. fairness of components) can be defined just as in Definition 7 by replacing each action transition $\stackrel{\alpha}{\rightarrow}$ and time step $\stackrel{1}{\rightarrow}$ with its counterpart in the component-oriented timed operational semantics, i.e. $\stackrel{\alpha}{\longmapsto}$ and $\stackrel{1}{\longmapsto}$. To keep things short, we do not report here the formal definition.

3 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. 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;
```

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.

3.1 Translating the Algorithm into PAFAS Processes

In our translation of the algorithm into PAFAS, we use essentially the same coding as given by Walker in [11]. 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 $\mathsf{B}_1(\mathit{false})$ is the set $\{b_1rf, b_1rt, b_1wf, b_1wt\}$ where b_1rf and b_1rt represent the actions of reading the values *false* and *true* from b_1, b_1wf and b_1wt represent, respectively, the writing of the values *false* and *true* into b_1. Let $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$ and $\mathbb{K} = \{1, 2\}$.

Definition 13. (*program variables*) Let $i \in \{1, 2\}$. We define the processes representing program variables as follows:

$$\mathsf{B}_i(\mathit{false}) = b_irf.\mathsf{B}_i(\mathit{false}) + (b_iwf.\mathsf{B}_i(\mathit{false}) + b_iwt.\mathsf{B}_i(\mathit{true}))$$
$$\mathsf{B}_i(\mathit{true}) = b_irt.\mathsf{B}_i(\mathit{true}) + (b_iwf.\mathsf{B}_i(\mathit{false}) + b_iwt.\mathsf{B}_i(\mathit{true}))$$
$$\mathsf{K}(i) = kri.\mathsf{K}(i) + (kw1.\mathsf{K}(1) + kw2.\mathsf{K}(2))$$

Let $B = \{b_irf, b_irt, b_iwf, b_iwt \mid i \in \{1, 2\}\} \cup \{kr1, kr2, kw1, kw2\}$ 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 $b_1, b_2 \in \mathbb{B}$, $k \in \mathbb{K}$ and using $\|$ as a shorthand for $\|_\emptyset$, we define $\mathsf{PV}(b_1, b_2, k) = (\mathsf{B}_1(b_1) \parallel \mathsf{B}_2(b_2)) \parallel \mathsf{K}(k)$.

Definition 14. (*the algorithm*) The processes $\mathtt{P}_1$ and $\mathtt{P}_2$ are represented by the following PAFAS processes; the actions $\mathtt{req}_i$ and $\mathtt{cs}_i$ have been added to indicate the request to enter and the execution of the critical section by the process $\mathtt{P}_i$.

$$P_1 = \mathtt{req}_1.b_1wt.P_{11} + \tau.P_1 \qquad P_2 = \mathtt{req}_2.b_2wt.P_{21} + \tau.P_2$$
$$P_{11} = b_2rf.P_{14} + b_2rt.P_{12} \qquad P_{21} = b_1rf.P_{24} + b_1rt.P_{22}$$
$$P_{12} = kr1.P_{11} + kr2.b_1wf.P_{13} \qquad P_{22} = kr2.P_{21} + kr1.b_2wf.P_{23}$$
$$P_{13} = kr1.b_1wt.P_{11} + kr2.P_{13} \qquad P_{23} = kr2.b_2wt.P_{21} + kr1.P_{23}$$
$$P_{14} = \mathtt{cs}_1.kw2.b_1wf.P_1 \qquad P_{24} = \mathtt{cs}_2.kw1.b_2wf.P_2$$

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

3.2 Liveness Property of Dekker's Algorithm

As discussed in the introduction, a mutex algorithm satisfies its liveness property if whenever at any point in any computation a process $\mathtt{P}_i$ requests the execution of its critical section, then, in any continuation of that computation, there is a point at which $\mathtt{P}_i$ will perform its critical section. We can expect this property to hold only under some fairness assumption; so for the formal property we want to check, we replace 'computation' by 'fair trace' (in one of our two interpretations). In other words, a mutex algorithm satisfies its *liveness property* if any occurrence of $\mathtt{req}_i$ in a fair trace is eventually followed by $\mathtt{cs}_i$, $i = 1, 2$. Due to our definition of fair trace, this amounts to checking that each non-Zeno timed execution sequence is live according to the following definition.

Definition 15. (*live execution sequences*) Let $P_0 \in \mathbb{P}_1$, $\lambda_0, \lambda_1, \ldots \in (\mathbb{A}_d \cup \{\tau\} \cup \{1\})$. A *timed execution sequence* γ from P_0 with $\gamma = P_0 \xrightarrow{1} Q_0 \xrightarrow{\lambda_0} Q_1 \xrightarrow{\lambda_1} \ldots$ $(\gamma = P_0 \overset{1}{\longmapsto} Q_0 \overset{\lambda_0}{\longmapsto} Q_1 \overset{\lambda_1}{\longmapsto} \ldots)$ is *not live* if there exists $j \in \mathbb{N}_0$ such that $\lambda_j = \mathtt{req}_i$ and $\mathtt{cs}_i$ is not performed in the execution sequence $Q_{j+1} \xrightarrow{\lambda_{j+1}} Q_{j+2} \xrightarrow{\lambda_{j+2}} \ldots$ $(Q_{j+1} \overset{\lambda_{j+1}}{\longmapsto} Q_{j+2} \overset{\lambda_{j+2}}{\longmapsto} \ldots$ respectively). Otherwise, we say that γ is live.

4 Fairness of Actions and Liveness

This section shows that fairness of actions is not sufficiently strong to ensure the liveness property. We present two fair traces with respect to fairness of actions, which violate the liveness property, i.e. two non-Zeno timed execution sequences in PAFAS (cf. Section 2.1) which are not live. We now describe how program variables and the processes $\mathtt{P}_1$ and $\mathtt{P}_2$ evolve by letting one time unit pass.

Definition 16. (*urgent program variables*) According to Definitions 13 and 6, *urgent program variables* can be defined as follows:

$$\underline{\mathsf{B}_i}(\mathit{false}) = \underline{b_i rf}.\mathsf{B}_i(\mathit{false}) + (\underline{b_i wf}.\mathsf{B}_i(\mathit{false}) + \underline{b_i wt}.\mathsf{B}_i(\mathit{true}))$$
$$\underline{\mathsf{B}_i}(\mathit{true}) = \underline{b_i rt}.\mathsf{B}_i(\mathit{true}) + (\underline{b_i wf}.\mathsf{B}_i(\mathit{false}) + \underline{b_i wt}.\mathsf{B}_i(\mathit{true}))$$
$$\underline{\mathsf{K}}(i) = \underline{kri}.\mathsf{K}(i) + (\underline{kw1}.\mathsf{K}(1) + \underline{kw2}.\mathsf{K}(2))$$

Let us denote with $\underline{\mathbb{B}} = \{\underline{\mathit{false}}, \underline{\mathit{true}}\}$ and with $\underline{\mathbb{K}} = \{\underline{1}, \underline{2}\}$. Then, given $b'_1, b'_2 \in \mathbb{B} \cup \underline{\mathbb{B}}$ and $k' \in \mathbb{K} \cup \underline{\mathbb{K}}$, we define $\mathsf{PV}(b'_1, b'_2, k') = ((B_1 \parallel B_2) \parallel K)$, where:

$$B_i = \begin{cases} \mathsf{B}_i(b) & \text{if } b'_i = b \in \mathbb{B} \\ \underline{\mathsf{B}_i}(b) & \text{if } b'_i = \underline{b} \in \underline{\mathbb{B}} \end{cases} \qquad K = \begin{cases} \mathsf{K}(k) & \text{if } k' = k \in \mathbb{K} \\ \underline{\mathsf{K}}(k) & \text{if } k' = \underline{k} \in \underline{\mathbb{K}} \end{cases}$$

As an example, we have that $\mathsf{PV}(\underline{\mathit{true}}, \mathit{false}, \underline{2}) = (\underline{\mathsf{B}_1}(\mathit{true}) \parallel \mathsf{B}_2(\mathit{false})) \parallel \underline{\mathsf{K}}(2)$.

The urgent versions of processes P_1 and P_2, denoted by $\underline{P}_1$ and $\underline{P}_2$ resp., are as in Definition 14 where initial actions are urgent. We use $\underline{P}_{ij}$ ($i = 1, 2$ and $j = 1, 2, 3, 4$) to denote the urgent versions of their derivatives (ex. $\underline{P}_{12} = \underline{kr1}.P_{11} + \underline{kr2}.b_1 wf.P_{13}$). As a consequence of the above definitions (and by the action-oriented operational semantics) we have that *Dekker* can let one time unit pass evolving into $\underline{\mathit{Dekker}} = ((\underline{P}_1 \parallel \underline{P}_2) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B]$. Our first example shows how an infinite τ-loop can result in the starvation of both processes.

Example 1. Let us consider the following timed computation from *Dekker*:

$$\begin{aligned} \mathit{Dekker} \xrightarrow{1} \underline{\mathit{Dekker}} = ((\underline{P}_1 \parallel \underline{P}_2) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] \quad & \xrightarrow{\mathtt{req}_1} \xrightarrow{\mathtt{req}_2} \xrightarrow{\tau^4} \\ P_0 = ((P_{11} \parallel b_2 wf.P_{23}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{1} \\ Q_0 = ((\underline{P}_{11} \parallel \underline{b_2 wf}.P_{23}) \parallel_B \mathsf{PV}(\mathit{true}, \underline{\mathit{true}}, 1))[\Phi_B] \quad & \xrightarrow{\tau^2} \\ P_0 = ((P_{11} \parallel \underline{b_2 wf}.P_{23}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{true}, 1))[\Phi_B] & \end{aligned}$$

Repeating the last three transitions, we get a non-Zeno timed execution sequence that is not live, i.e. *Dekker* can perform a fair trace $\mathit{Dekker} \xrightarrow{\mathtt{req}_1\, \mathtt{req}_2\, \tau^4} P_0 \xrightarrow{\tau^2} P_0 \xrightarrow{\tau^2} P_0 \ldots$ that violates liveness since no process will ever enter its critical section. Intuitively speaking, once in P_0, repeated reading of variables b_2 and k blocks indefinitely $\mathtt{P}_2$ which will never set its request variable b_2 to false. On the other hand, $\mathtt{P}_1$ cannot enter its critical section and, hence, cannot proceed until the value of b_2 is true. Thus, both processes are stuck. The

next example shows a different kind of computation which also causes a violation of liveness; along such a computation, one process is stuck while the other repeatedly executes its critical section. Consider the following computation:

$$\begin{array}{l} \mathit{Dekker} \xrightarrow{1} \underline{\mathit{Dekker}} = ((\underline{P_1} \parallel \underline{P_2}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] \xrightarrow{\mathtt{req}_1\ \mathtt{req}_2\ \tau^2\ \mathtt{cs}_1\ \tau} \\ P_0 = ((b_1\, wf.P_1 \parallel b_2\, wt.P_{21}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{false}, 2))[\Phi_B] \xrightarrow{1} \\ ((\underline{b_1\, wf}.P_1 \parallel \underline{b_2\, wt}.P_{21}) \parallel_B \mathsf{PV}(\underline{\mathit{true}}, \underline{\mathit{false}}, 2))[\Phi_B] \xrightarrow{\tau\ \mathtt{req}_1\ \tau^2\ \mathtt{cs}_1\ \tau} \\ P_0 = ((b_1\, wf.P_1 \parallel b_2\, wt.P_{21}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{false}, 2))[\Phi_B] \end{array}$$

Again, the trace performed in $\mathit{Dekker} \xrightarrow{\mathtt{req}_1\ \mathtt{req}_2\ \tau^2\ \mathtt{cs}_1\ \tau} P_0 \xrightarrow{\tau\ \mathtt{req}_1\ \tau^2\ \mathtt{cs}_1\ \tau} P_0 \ldots$ is fair but violates liveness since $\mathtt{P}_2$ never enters its critical section. Here, $\mathtt{P}_1$ repeatedly executes its critical section, again preventing $\mathtt{P}_2$ to set its request variable b_2 to *true*. As a consequence, $\mathtt{P}_2$ cannot enter its critical section even if the value of turn variable k is two.

5 Fairness of Components and Liveness

This section proves that any fair trace of *Dekker* according to fairness of components satisfies the liveness property. We present three ideas to reduce the number of states we have to deal with.

5.1 Permanently Lazy Components

The state space of a process in PAFAS$^{\mathrm{C}}$ is considerably larger than in an untimed process algebra because process components switch from lazy to urgent. We can achieve a considerable reduction, if we prevent this by declaring some components as permanently lazy. As an application, we regard the three program variables as one component of *Dekker*; declaring it as permanently lazy results in a process denoted by $\mathit{Dekker}[\mathsf{PV}]$. A non-Zeno timed execution sequence of the original process can be simulated by one of the new process. Thus, instead of proving that all non-Zeno timed execution sequences of *Dekker* are live, it is sufficient to prove that all non-Zeno timed execution sequences of $\mathit{Dekker}[\mathsf{PV}]$ are live. We have also a good intuitive reason to request it to be true. Since fairness is required for all components, a program variable can, intuitively speaking, enforce to be read or written - provided there is always some component that could do so. But our intuition for variables is that they are passive, that we really only want fairness towards $\mathtt{P}_1$ and $\mathtt{P}_2$. Assuming this kind of fairness is indeed enough. We now extend PAFAS$^{\mathrm{C}}$ with a new operator, which can only be applied to a top-level component.

Definition 17. (*permanently lazy processes*) Given $P \in \tilde{\mathbb{P}}_1$, we define the *permanently lazy version* of P, written $[P]$, to be the process with the same syntactical structure of P (and, hence, the same functional behaviour) but which permanently ignores the passage of time. The timed operational semantics of $[P]$ can be defined by the following rules:

$$\textsc{Act}_L\ \frac{P \stackrel{\alpha}{\longmapsto} P'}{[P] \stackrel{\alpha}{\longmapsto} [P']} \qquad \textsc{Time}_L\ \frac{}{[P] \stackrel{1}{\longmapsto} [P]}$$

The set $\tilde{\mathbb{P}}_{\ell 1}$ of initial processes with one permanently lazy top-level component is generated by:

$$S ::= P \parallel_A [P] \mid S[\Phi]$$

where $P \in \tilde{\mathbb{P}}_1$, $A \subseteq \mathbb{A}$ (possibly infinite) and Φ is a general relabelling function. Similarly, the set $\tilde{\mathbb{P}}_\ell$ of *(general) processes with one permanently lazy top-level component* is generated by the following grammar:

$$R ::= Q \parallel_A [P] \mid R[\Phi]$$

where $Q \in \tilde{\mathbb{P}}_c$, $P \in \tilde{\mathbb{P}}_1$, $A \subseteq \mathbb{A}$ (possibly infinite) and Φ is a general relabelling function.

We define the operational semantics for processes with one permanently lazy top-level component.

Definition 18. (*Functional operational semantics*) The following SOS-rules define the transition relations $\stackrel{\alpha}{\longmapsto} \subseteq (\tilde{\mathbb{P}}_\ell \times \tilde{\mathbb{P}}_\ell)$ for $\alpha \in \mathbb{A}_\tau$, the *action transitions*.

$$\textsc{Lpar}_{a1}\ \frac{\alpha \notin A,\ Q \stackrel{\alpha}{\longmapsto} Q'}{Q \parallel_A [P] \stackrel{\alpha}{\longmapsto} \mathsf{clean}(Q' \parallel_A [P])} \qquad \textsc{Lpar}_{a2}\ \frac{\alpha \notin A, P \longmapsto_\alpha P'}{Q \parallel_A [P] \stackrel{\alpha}{\longmapsto} \mathsf{clean}(Q \parallel_A [P'])}$$

$$\textsc{Lsynch}_a\ \frac{\alpha \in A,\ Q \stackrel{\alpha}{\longmapsto} Q', P \longmapsto_\alpha P'}{Q \parallel_A [P] \longmapsto_\alpha \mathsf{clean}(Q' \parallel_A [P'])} \qquad \textsc{Lrel}_a\ \frac{R \stackrel{\alpha}{\longmapsto} R'}{R[\Phi] \longmapsto_{\Phi(\alpha)} R'[\Phi]}$$

where $\mathsf{clean}(Q \parallel_A [P]) = \mathsf{clean}(Q, A') \parallel_A [P]$ and $A' = (\mathcal{A}(Q) \backslash \mathcal{A}(P)) \cap A$.

Definition 19. (*time step*) For $S \in \tilde{\mathbb{P}}_{\ell 1}$, we write that $S \stackrel{1}{\longmapsto} R$ when $R = \mathsf{urgent}(S)$ where function $\mathsf{urgent}(S)$ is defined as follows:

Par: $\mathsf{urgent}(P_1 \parallel_B [P_2]) = \mathsf{urgent}(P_1, A') \parallel_B [P_2]$ where $A' = (\mathcal{A}(P_1) \backslash \mathcal{A}(P_2)) \cap A$
Rel: $\mathsf{urgent}(S[\Phi]) = \mathsf{urgent}(S)[\Phi]$

Definition 20. Let $Q \in \tilde{\mathbb{P}}$ and $R \in \tilde{\mathbb{P}}_\ell$. We write that $Q \preceq R$ if either $Q = Q_1 \parallel_A Q_2$ and $R = Q_1 \parallel_A [\mathsf{unmark}(Q_2)]$ or $Q = Q_1[\Phi]$ and $R = R_1[\Phi]$ with $Q_1 \preceq R_1$.

Proposition 1. Let $P \in \tilde{\mathbb{P}}_1$, $S \in \tilde{\mathbb{P}}_{\ell 1}$ with $P \preceq S$ and $v \in (\mathbb{A}_\tau)^*$. Then $P \stackrel{1}{\longmapsto} Q \stackrel{v}{\longmapsto} P' \in \tilde{\mathbb{P}}_1$ implies $S \stackrel{1}{\longmapsto} R \stackrel{v}{\longmapsto} S'$ with $P' \preceq S'$ (S simulates each non-Zeno timed execution sequence of P).

This proposition states that all non-Zeno timed execution sequences of *Dekker* can be simulated by non-Zeno timed execution sequences of *Dekker*[PV].

5.2 F-Steps

We can group the transitions of a non-Zeno timed execution sequence into infinitely many steps of the form $S \stackrel{1}{\longmapsto} R \stackrel{v}{\longmapsto} S'$, where $v \in (\mathbb{A}_\tau)^*$ and S' is the next process to perform a time step. Such a step is minimal in a sense, if S' is the first process in the transition sequence $R \stackrel{v}{\longmapsto} S'$ that could perform a time step, i.e. the first initial process. We call such minimal steps f-steps and the processes reachable by them fair-reachable. We will show in this subsection that we only have to consider timed execution sequences built from infinitely such f-steps.

Definition 21. (*f-executions*) A transition sequence $S \stackrel{1}{\longmapsto} R \stackrel{v}{\longmapsto} S'$ with $S, S' \in \mathbb{P}_{\ell 1}$ and $v \in (\mathbb{A}_\tau)^*$ is an *f-step* if S' is the only initial process in the transition sequence $R \stackrel{v}{\longmapsto} S'$ (allowing $R = S'$ if v is the empty sequence). An *f-execution* from $S_0 \in \mathbb{P}_{\ell 1}$ is any infinite sequence of f-steps of the form: $\gamma = S_0 \stackrel{1}{\longmapsto} R_0 \stackrel{v_0}{\longmapsto} S_1 \stackrel{1}{\longmapsto} R_1 \stackrel{v_1}{\longmapsto} S_2 \ldots$ We call the processes $S_1, S_2, \ldots$ *fair-reachable* from S_0.

F-executions are special non-Zeno timed execution sequences. To show that checking them for liveness suffices, we need the following proposition.

Proposition 2. For each non-Zeno timed execution sequence from $S_0 \in \mathbb{P}_{\ell 1}$, $\gamma = S_0 \stackrel{1}{\longmapsto} R_0 \stackrel{v_0}{\longmapsto} S_1 \stackrel{1}{\longmapsto} R_1 \stackrel{v_1}{\longmapsto} S_2 \ldots$ there exists a corresponding f-execution $\gamma' = S'_0 \stackrel{1}{\longmapsto} R'_0 \stackrel{v'_0}{\longmapsto} S'_1 \stackrel{1}{\longmapsto} R'_1 \stackrel{v'_1}{\longmapsto} S'_2 \ldots$, where $S'_0 = S_0$, $v_0 v_1 \ldots = v'_0 v'_1 \ldots$ and each step $S'_i \stackrel{1}{\longmapsto} R'_i \stackrel{v'_i}{\longmapsto} S'_{i+1}$ is minimal.

5.3 Symmetry of Fair-Reachable Processes

Half of the processes which are fair-reachable from *Dekker*[PV] are denoted by $D_1, \ldots, D_{47}$; see Table 1 and [4] for a full list of the processes. We also consider all possible symmetries and use S_y to denote the process which is symmetric to D_y with respect to the local state of $\mathtt{P}_1$ and $\mathtt{P}_2$ and the value of the variables b_1, b_2 and k. For each $y \in [1, 47]$, $S_y = \mathcal{S}(D_y)$ where function $\mathcal{S}(\_)$ on processes is given below. Moreover, $\mathcal{S}(S_y) = D_y$ for any y.

Table 1. Fair-Reachable Processes

$D_1 = ((b_1 wt.P_{11} \parallel b_2 wt.P_{21}) \parallel_B [\mathsf{PV}(false, false, 1)])[\Phi_B]$
$\vdots$
$D_{23} = ((b_1 wf.P_1 \parallel P_{21}) \parallel_B [\mathsf{PV}(true, true, 2)])[\Phi_B]$
$\vdots$
$D_{47} = (P_{13} \parallel b_2 wt.P_{21}) \parallel_B [\mathsf{PV}(false, false, 2)])[\Phi_B]$

Definition 22. *(symmetric processes)* Let $P_1, P_{11}, \ldots P_{14}$, $P_2, P_{21}, \ldots P_{24}$ be processes as given in Definition 14. Let moreover $x \in [1, 4]$ and $\{i, j\} = \{1, 2\}$. Then:

$$\mathcal{S}(P_i) = P_j \qquad \mathcal{S}(P_{ix}) = P_{jx}$$
$$\mathcal{S}(b_i wt.P_{i1}) = b_j wt.P_{j1} \qquad \mathcal{S}(b_i wf.P_{i3}) = b_j wf.P_{j3}$$
$$\mathcal{S}(kwj.b_i wf.P_i) = kwi.b_j wf.P_j \qquad \mathcal{S}(b_i wf.P_i) = b_j wf.P_j$$

Now, let $b_1, b_2 \in \mathbb{B}$, $k \in \mathbb{K}$ and $S = ((S_1 \parallel S_2) \parallel_B [\mathsf{PV}(b_1, b_2, k)])[\Phi]$ be action-reachable from $Dekker[\mathsf{PV}]$. We can define the symmetric process of S as follows:

$$\mathcal{S}(S) = ((\mathcal{S}(S_2) \parallel \mathcal{S}(S_1)) \parallel_B [\mathsf{PV}(b_2, b_1, (k \mod 2) + 1)])[\Phi_B]$$

We say that two processes S and S' action-reachable from $Dekker[\mathsf{PV}]$ are symmetric, written $S \approx S'$, if either $S' = \mathcal{S}(S)$ or $S = \mathcal{S}(S')$.

Definition 23. *(symmetric sequences of actions)* Given $v \in (\mathbb{A}_d \cup \{\tau\})^*$, the string $\mathcal{S}(v)$ is defined, by induction on the length of v, as follows: $\mathcal{S}(\varepsilon) = \varepsilon$, $\mathcal{S}(\tau\, v') = \tau\, \mathcal{S}(v')$, $\mathcal{S}(\mathtt{req}_i\, v') = \mathtt{req}_j\, v'$ and $\mathcal{S}(\mathtt{cs}_i\, v') = \mathtt{cs}_j\, v'$ where $i, j \in \{1, 2\}$

Proposition 3 states that symmetric processes have symmetric behaviours: they perform symmetric f-steps and evolve into processes which are still symmetric.

Proposition 3. Let $S \approx S'$ and $v \in (\mathbb{A}_d \cup \{\tau\})^*$. Then: $S' \stackrel{1}{\longmapsto} R' \stackrel{v}{\longmapsto} S'_0 \in \tilde{\mathbb{P}}_{\ell 1}$ implies $S \stackrel{1}{\longmapsto} R \stackrel{\mathcal{S}(v)}{\longmapsto} S_0 \in \tilde{\mathbb{P}}_{\ell 1}$ with $S_0 \approx S'_0$, and one is an f-step if and only if the other one is.

Let $D_0 = Dekker[\mathsf{PV}]$, $\mathtt{D} = \{D_0, \ldots D_{47}\}$ and $\mathtt{S} = \{S_0, \ldots S_{47}\}$. The following proposition shows that all processes fair-reachable from $Dekker[\mathsf{PV}]$ are in $\mathtt{D} \cup \mathtt{S}$.

Proposition 4. Let $S \in \mathtt{D} \cup \mathtt{S}$ and $v \in (\mathbb{A}_d \cup \{\tau\})^*$. $S \stackrel{1}{\longmapsto} R \stackrel{v}{\longmapsto} S' \in \tilde{\mathbb{P}}_{\ell 1}$ implies $S' \in \mathtt{D} \cup \mathtt{S}$.

5.4 Progressing Processes

We distinguish terms in $\mathtt{D} \cup \mathtt{S}$ depending on how many processes are waiting to perform their critical section, i.e. depending on how many actions $\mathtt{cs}_i$ are still pending. The action $\mathtt{cs}_i$ is *pending*, for a given $S \in \mathtt{D} \cup \mathtt{S}$, if there exist sequences of basic actions $v, w \in (\mathbb{A}_d \cup \{\tau\})^*$ such that $Dekker[\mathsf{PV}] \stackrel{v\, \mathtt{req}_i\, w}{\longmapsto} S$ and $\mathtt{cs}_i \notin w$. Trivially, each process may have at most two pending actions and hence $\mathtt{D} \cup \mathtt{S} = \mathtt{R}_1 \cup \mathtt{R}_2 \cup \mathtt{R}_{1,2}$, where $\mathtt{R}_1$ ($\mathtt{R}_2$) is the set of fair-reachable states with only $\mathtt{cs}_1$ ($\mathtt{cs}_2$, resp.) pending and $\mathtt{R}_{1,2}$ is the set of fair-reachable states with both $\mathtt{cs}_1$ and $\mathtt{cs}_2$ pending.

We may check if a given fair-reachable process S belongs to $\mathtt{R}_1$, $\mathtt{R}_2$ or $\mathtt{R}_{1,2}$ by considering its syntactical structure and, in particular, the local states of $\mathtt{P}_1$ and $\mathtt{P}_2$ in S. In detail, we distinguish the following subsets of fair-reachable processes: $\mathtt{D}_1 = \mathtt{D} \cap \mathtt{R}_1 = \{D_2, D_5, D_9, D_{14}, D_{22}, D_{32}\}$, $\mathtt{D}_2 = \mathtt{D} \cap \mathtt{R}_2 = \{D_3, D_7, D_{11}, D_{12}, D_{16}, D_{21}, D_{23}, D_{29}, D_{30}, D_{31}, D_{33}, \ldots, D_{36}, D_{40}\}$ and $\mathtt{D}_{1,2} = \mathtt{D} \cap \mathtt{R}_{1,2} = \{D_1, D_4, D_6, D_8, D_{10}, D_{18}, D_{19}, D_{20}, D_{24}, \ldots, D_{28}, D_{37}, D_{38}, D_{39}, D_{41}, D_{42}, \ldots, D_{47}\}$. Processes D_0, D_{13}, D_{15} and D_{17} have no pending sections. Since $S \in \mathtt{R}_1$, $S \in \mathtt{R}_2$ and $S \in \mathtt{R}_{1,2}$ imply $\mathcal{S}(S) \in \mathtt{R}_2$, $\mathcal{S}(S) \in \mathtt{R}_1$ and $\mathcal{S}(S) \in \mathtt{R}_{1,2}$, respectively, we also have: $\mathtt{S}_1 = \mathtt{S} \cap \mathtt{R}_1 = \mathcal{S}(\mathtt{D}_2)$, $\mathtt{S}_2 = \mathtt{S} \cap \mathtt{R}_2 = \mathcal{S}(\mathtt{D}_1)$ and $\mathtt{S}_{1,2} = \mathtt{S} \cap \mathtt{R}_{1,2} = \mathcal{S}(\mathtt{D}_{1,2})$. Finally, S_0, S_{13}, S_{15} and S_{17} have no pending actions.

Definition 24. (*progressing processes*) We say that a string $v = \alpha_1 \dots \alpha_n \in (\mathbb{A}_d \cup \{\tau\})^*$ contains the action cs_i ($i \in \{1,2\}$), writen $\mathsf{cs}_i \in v$, if $\alpha_j = \mathsf{cs}_i$ for some $j \in [1,n]$. Trivially, $\mathsf{cs}_1, \mathsf{cs}_2 \in v$ if both $\mathsf{cs}_1 \in v$ and $\mathsf{cs}_2 \in v$.

A given $S \in \mathtt{D} \cup \mathtt{S}$ implies the execution of the action cs_i, denoted by $S \rhd \mathsf{cs}_i$, if each f-execution from S contains the action cs_i; $S \rhd \mathsf{cs}_1, \mathsf{cs}_2$ if both $S \rhd \mathsf{cs}_1$ and $S \rhd \mathsf{cs}_2$. Finally, we say that $S \in \mathtt{R}_1$ (symmetrically for $S \in \mathtt{R}_2$ and $S \in \mathtt{R}_{1,2}$) is *making progress (progressing)* if $S \rhd \mathsf{cs}_1$ ($S \rhd \mathsf{cs}_2$, $S \rhd \mathsf{cs}_1, \mathsf{cs}_2$, respectively).

Proposition 5. Let $S, S' \in \mathtt{D} \cup \mathtt{S}$ with $S \approx S'$. Then: (*i*) $S \rhd \mathsf{cs}_i$ implies $S' \rhd \mathsf{cs}_j$ with $\{i,j\} \in \{1,2\}$; (*ii*) if S is making progress then also S' is making progress.

Now, we prove that all processes in $\mathtt{D}$ and hence, by Proposition 5-(*ii*), all processes in $\mathtt{D} \cup \mathtt{S}$ are progressing. We need the following statement.

Proposition 6. A given $D_y \rhd \mathsf{cs}_i$ if for any f-step from D_y of the form $D_y \stackrel{1}{\longmapsto} R \stackrel{v}{\longmapsto} S \in \tilde{\mathbb{P}}_{\ell 1}$ we have either $\mathsf{cs}_i \in v$ or $S \rhd \mathsf{cs}_i$.

Iterative application of Proposition 6 allows us to state which processes can perform specific actions.

Lemma 1. All processes in $\mathtt{D} \cup \mathtt{S}$ are making progress.

Proposition 7. Each f-execution from *Dekker*[PV] is live.

As an immediate consequence of the relationships between fair traces of *Dekker* and f-executions of *Dekker*[PV], we have the main result of this section:

Theorem 1. Each fair trace of *Dekker* satisfies the liveness property.

References

1. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. In *Proceedings of ACM Transaction on Programming Languages and Systems*, **15**, 1993.
2. F. Corradini, M.R. Di Berardini, and W. Vogler. Fairness of Actions in System Computations. To appear in *Acta Informatica* Extended abstract: Relating Fairness and Timing in Process Algebras. Proc. of *Concur'03*, Lect. Notes Comp. Sci. 2761, pp. 446-460, 2003.
3. F. Corradini, M.R. Di Berardini, and W. Vogler. Fairness of Components in System Computations. *Theoretical Computer Science* **356**, pp. 291-324, 2006.
4. F. Corradini, M.R. Di Berardini, and W. Vogler. Checking a Mutex Algorithm in a Process Algebra with Fairness. Full Version, Avalilable on line at http://www.cs.unicam.it/docenti/flavio.corradini
5. F. Corradini, W. Vogler, and L. Jenner. Comparing the Worst-Case Efficiency of Asynchronous Systems with PAFAS. *Acta Informatica* **38**, pp. 735-792, 2002.
6. G. Costa, C. Stirling. A Fair Calculus of Communicating Systems. *Acta Informatica* **21**, pp. 417-441, 1984.
7. G. Costa, C. Stirling. Weak and Strong Fairness in CCS. *Information and Computation* **73**, pp. 207-244, 1987.

8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
9. R. Milner. *Communication and Concurrency*. International series in computer science, Prentice Hall International, 1989.
10. J.L. Peterson, A. Silberschatz . *Operating Systems Concepts*. Addison Wiley, 1985.
11. D.J. Walker. Automated Analysis of Mutual Exclusion algorithms using CCS. *Formal Aspects of Computing* **1**, pp. 273-292, 1989.