Acta Informatica (2009) 46:209–235
DOI 10.1007/s00236-009-0092-9

ORIGINAL ARTICLE

Liveness of a mutex algorithm in a fair process algebra

Flavio Corradini · Maria Rita Di Berardini · Walter Vogler

Received: 3 March 2008 / Accepted: 4 February 2009 / Published online: 24 March 2009

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 (Form Asp Comput 1:273–292, 1989) 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 [15] who aimed at automatically verifying six mutual exclusion algorithms—including Dekker's. Walker translated the algorithms into the process algebra CCS [13] 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 [10], which is not so

An extended abstract appeared under the title 'Checking a Mutex Algorithm in a Process Algebra with Fairness' in C. Baier and H. Hermanns, eds., CONCUR 2006, vol. 4137 of Lecture Notes in Computer Science, pp. 142–157, Bonn, Germany, 2006. Springer.

F. Corradini · M. R. Di Berardini
Dipartimento di Matematica e Informatica, Università di Camerino, Camerino, Italy
e-mail: flavio.corradini@unicam.it

M. R. Di Berardini
e-mail: mariarita.diberardini@unicam.it

W. Vogler (✉)
Institut für Informatik, Universität Augsburg, Augsburg, Germany
e-mail: vogler@informatik.uni-augsburg.de

easy to do in a process algebraic setting; with respect to the verification of liveness, Walker was less successful.

Costa and Stirling [8,9] 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, i.e. in all interesting cases. In [3,4], we have defined fair runs in the spirit of Costa and Stirling and characterised them in the timed process algebra PAFAS 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, and shows that PAFAS can express fairness well enough to study relevant issues.

Attempting the verification of the liveness property, Walker used the following version in [15]—which could be expressed as a modal mu-calculus formula and checked with the Concurrency Workbench:

> 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.

Observe the following: on the one hand, PAFAS is a timed process algebra where timing is added in the form of maximal delays for actions (for such an approach in a different setting, also cf. the part on asynchronous algorithms in [12]). This timing does not change the functionality (which actions are performed) and allows to compare the performance of asynchronous systems; see [7] for a respective testing-based faster-than precongruence and [2,6] for applications. On the other hand, in this paper, timing is only used to have a simple description of fair runs; on this basis, we will verify the liveness of Dekker's algorithm; the latter does not take timing into account and neither are we concerned with its performance.

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 synchronisation partner synchronises its 'part' of the action with some other process. 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 the subclass of such timed computations where each time step occurs as soon as possible 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. Note that this second result involves a small extension to PAFAS.

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. A third result allows us to check liveness of a generic fair-reachable derivative to deduce the same property of the symmetric one.

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 [7] useful to be embedded within an automatic tool for the verification based on fairness.

The rest of the paper is organised as follows. The next section recalls PAFAS, its functional and temporal operational semantics and the fairness notions we are interested in (actions and components). Section 3 describes Dekker's mutual exclusion algorithm and its description within the PAFAS process algebra, and it introduces the liveness property we consider. Section 4 shows that fairness of actions is not suitable for liveness while Sect. 5 proves the main result of this paper, namely, that any computation from *Dekker* that is fair according to fairness of components satisfies liveness. The proof makes use of the three results described above; further details of this proof can be found in the technical report [5].

2 Fairness and PAFAS

In this section we recall PAFAS, its timed behaviour and the fairness notions we consider in the rest of the paper, namely fairness of actions and components. Instead of using the very involved direct formalisations of fairness in the spirit of [8,9], 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

PAFAS is a CCS-like process description language [13] (with *TCSP*-like parallel composition), where basic actions are atomic and instantaneous but have associated a time bound interpreted as a maximal time delay for their execution. As explained in [7], these upper time bounds (which are either 0 or 1, for simplicity) are suitable for evaluating the performance of asynchronous systems. Moreover, time bounds do not influence functionality (which actions are performed); so compared to CCS, also PAFAS treats the full functionality of asynchronous systems.

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 $\alpha \notin A$.

We assume that time elapses in a discrete way (PAFAS is not time domain dependent, meaning that the choice of discrete or continuous time makes no difference for the testing-based semantics of asynchronous systems, see [7] for more details). Thus, an action prefixed process $a.P$ can either do action a and become process P (as usual in CCS) or can let one time step pass and become $\underline{a}.P$; $\underline{a}$ is called *urgent* a, and $\underline{a}.P$ as a stand-alone process cannot let time pass, but can only do a to become P.

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:

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

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 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 rec x-operator; such a Q in $\tilde{\mathbb{P}}$ and $\tilde{\mathbb{P}}_1$ is simply called *process* and *initial process* resp., and their sets are denoted by $\mathbb{P}$ and $\mathbb{P}_1$ resp.[1]

[1] In [7], 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.

A brief description of the (PAFAS) operators now follows. The Nil-process nil cannot perform any action, but may let time pass without limit. A trailing nil will often be omitted, so e.g. $a.b + c$ abbreviates $a.b.\mathsf{nil} + c.\mathsf{nil}$. $Q_1 + Q_2$ models the choice between two conflicting processes Q_1 and Q_2. $Q_1 \|_A Q_2$ is the parallel composition of two processes Q_1 and Q_2 that run in parallel and have to synchronise on all actions from A; this synchronisation discipline is inspired from TCSP. $Q[\Phi]$ behaves as Q but with the actions changed according to Φ. $\mathsf{rec}\, x.Q$ models a recursive definition.

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, $\mathsf{A}(Q, A)$ denotes the set of the activated (or enabled) actions of Q when the environment (in a parallel context) prevents the actions in A.

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

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

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

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

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

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

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

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

The set A represents the actions restricted upon. Thus, $\mathsf{A}(\alpha.P, A) = \mathsf{A}(\underline{\alpha}.P, A) = \emptyset$ if $\alpha \in A$ and $\mathsf{A}(\alpha.P, A) = \mathsf{A}(\underline{\alpha}.P, A) = \{\alpha\}$, if $\alpha \notin A$. A nondeterministic process can perform all the actions that its alternative components can perform minus the restricted ones. Parallel composition increases the prevented set: $\mathsf{A}(Q_1 \|_B Q_2, A)$ includes the actions that Q_1 can perform when we prevent all actions in A plus all actions in B that Q_2 cannot perform, and it includes the analogous actions of Q_2. The other rules are as expected.

A significant subset of the activated actions is the set of urgent ones. These are activated actions that cannot let time pass.

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

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

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

The operational semantics exploits two functions on terms in $\tilde{\mathbb{P}}$: $\mathsf{clean}(\_)$ and $\mathsf{unmark}(\_)$. 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 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 $Q \in \tilde{\mathbb{P}}$. Both functions can be defined, as follows, by induction on the process structure.

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 follows:

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 $A' = (\mathsf{U}(Q_1) \backslash \mathsf{U}(Q_2)) \cap B$ and $A'' = (\mathsf{U}(Q_2) \backslash \mathsf{U}(Q_1)) \cap B$

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

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

Definition 5 (*cleaning all urgencies*) Let Q be a $\tilde{\mathbb{P}}$ term. Then $\mathsf{unmark}(Q)$ is the term obtained by replacing each $\underline{\alpha}$ by α.

2.1.1 The functional behaviour of PAFAS process

The transitional semantics describing the functional behaviour of PAFAS processes indicates which basic actions they can perform. Timing can be disregarded: when an action is performed, one cannot see whether it was urgent or not, i.e. $\underline{\alpha}.P \xrightarrow{\alpha} P$; on the other hand, component $\alpha.P$ has to act *within* time 1, i.e. it can also act immediately, giving $\alpha.P \xrightarrow{\alpha} P$.

Definition 6 (*Functional operational semantics*) The following SOS-rules define the transition relations $\xrightarrow{\alpha} \subseteq (\tilde{\mathbb{P}} \times \tilde{\mathbb{P}})$ for $\alpha \in \mathbb{A}_\tau$, the *action transitions*.

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.

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

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

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

Additionally, there are symmetric rules for PAR_{a1} and 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 $P_0 \xrightarrow{\alpha_0} P_1 \xrightarrow{\alpha_1} \cdots$ of action transitions, possibly ending with a process P_n.

When in the rules for parallel composition one component changes, this changes the context for the other component such that some urgent actions might be no longer urgent or get disabled. For the necessary changes to the marking, clean is called upon as announced above. E.g. in $(\underline{a}.\mathsf{nil}\|_\emptyset\underline{b}.\mathsf{nil}) \parallel_{\{a,b\}} (\underline{a}.\mathsf{nil} + \underline{b}.\mathsf{nil} + \underline{c}.a.b.\mathsf{nil}) \xrightarrow{c} (a.\mathsf{nil}\|_\emptyset b.\mathsf{nil}) \parallel_{\{a,b\}} (a.b.\mathsf{nil})$, both actions a and b lose their urgency: a can be delayed, since the third component only offers a new non-urgent synchronisation on a, and b is no longer enabled. In earlier versions of PAFAS, we had the step $\xrightarrow{c}$ going to $(\underline{a}.\mathsf{nil}\|_\emptyset\underline{b}.\mathsf{nil}) \parallel_{\{a,b\}} (a.b.\mathsf{nil})$; for the reasons just given, the latter process could perform a time step. Using clean, we avoid the prefixes $\underline{a}$ and $\underline{b}$, which might give the wrong impression that there are actions that have to be performed before the next time step. This also gives a closer relation between timed runs and fair traces, see the comment after Definition 8.

The use of unmark in rule REC_a has to be contrasted with the temporal behaviour defined next. Consider an initial process P; after a time-step, the recursive term $\mathsf{rec}\,x.P$ evolves to $\mathsf{rec}\,x.\mathsf{urgent}(P)$ (see Definition 7 below). Since occurrences of x in P are guarded, each x stands for a process which is not enabled yet and cannot have urgent actions; thus, these recursive calls in $\mathsf{rec}\,x.\mathsf{urgent}(P)$ refer to P and not to $\mathsf{urgent}(P)$, which explains the substitution in rule REC_a of Definition 6, which in turn shows the use of unmark; cf. the example at the end of Sect. 2.1.2.

2.1.2 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 7 (*time step, timed execution sequences*) For $P \in \tilde{\mathbb{P}}_1$, we write $P \xrightarrow{1} Q$ when $Q = \mathsf{urgent}(P)$, where $\mathsf{urgent}(P)$ (often also written $\underline{P}$) abbreviates $\mathsf{urgent}(P, \emptyset)$ and $\mathsf{urgent}(P, A)$ is defined as follows:

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 $A' = (\mathsf{A}(P_1)\backslash\mathsf{A}(P_2)) \cap B$ and $A'' = (\mathsf{A}(P_2)\backslash\mathsf{A}(P_1)) \cap B$

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 $\gamma = P_0 \xrightarrow{1} Q_0 \xrightarrow{\lambda_0} Q_1 \xrightarrow{\lambda_1} \ldots$ 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, which in a setting with discrete time means exactly that it ends with infinitely many action transitions without a time step.

Example 1 As an example for the definitions given so far, consider $P = (R \parallel_\emptyset W) \parallel_{\{r,w\}} V$, where $V = \mathsf{rec}\, x.(r.x + w.x)$ models a process variable, $R = \mathsf{rec}\, x.\, r.x$ and $W = \mathsf{rec}\ x.w.x$ model the activities of reading and writing the process variable, respectively. According to our operational semantics, we have $V \xrightarrow{1} \underline{V} = \mathsf{rec}\, x.\ (\underline{r}.x + \underline{w}.x) \xrightarrow{r} V$; observe that, intuitively speaking, each occurrence of x is replaced by $V = \mathsf{rec}\, x.\, \mathsf{unmark}(\underline{r}.x + \underline{w}.x)$ before the second transition. Furthermore, we have:

$$P \xrightarrow{1} (\underline{R} \parallel_\emptyset \underline{W}) \parallel_{\{r,w\}} \underline{V} = ((\mathsf{rec}\, x.\ \underline{r}.x) \parallel_\emptyset (\mathsf{rec}\, x.\ \underline{w}.x)) \parallel_{\{r,w\}} \mathsf{rec}\, x.\ (\underline{r}.x + \underline{w}.x) \xrightarrow{r} P$$

To explain the first transition, we notice that initial process P can synchronise either on action r or on action w. Both actions are activated so that they become urgent after one time unit. The second transition models the execution of action r by synchronising $\underline{R}$ and $\underline{V}$. These processes evolve into R and V, respectively. As a side effect, the urgent w looses its urgency since its synchronisation partner V offers a new, non-urgent synchronisation. At this stage, it is function clean in Definition 6 that operates this change.

2.1.3 Fairness of actions and timing

Intuitively, if an enabled action has to be performed within time 1, then before the second time unit passes it will be performed or disabled by a conflicting action, i.e. weak fairness is guaranteed. In more technical terms: if an action becomes enabled in a timed execution sequence of PAFAS and it is never performed or disabled, then it becomes urgent after the next time step; now any further time step is impossible, i.e. we have a Zeno run. According to this idea, we have proved in [4] a characterisation for fair traces with non-Zeno runs on the basis of an intuitive, but very complex definition of fair traces in the spirit of [8,9]. Here, we use this characterisation as definition, and define a trace as (weakly) fair (w.r.t. fairness of actions) if there exists a corresponding timed execution sequence with infinitely many time steps (see [4] for more details).

Definition 8 (*fair traces*) Let $P_0 \in \mathbb{P}_1$ and $\alpha_0, \alpha_1, \alpha_2, \ldots \in \mathbb{A}_\tau$. A trace $\alpha_0\alpha_1\alpha_2 \ldots$ 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. More in detail:

1. A finite trace $\alpha_0\alpha_1\alpha_2 \ldots \alpha_{n-1}$ 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 $v_0\, v_1 \ldots v_{m-1} = \alpha_0\, \alpha_1 \ldots \alpha_{n-1}$;

[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 [4]).

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 $v_0\, v_1 \ldots\, v_m\, \ldots = \alpha_0\, \alpha_1\, \ldots\, \alpha_i\, \ldots$.

It is not so difficult to see from Definition 6, that in both cases we have $P_0 \xrightarrow{v_0} \xrightarrow{v_1} \ldots$ (i.e. $P_0 \xrightarrow{\alpha_0} \xrightarrow{\alpha_1} \ldots$); the intuitive explanation is that our delays are only upper time bounds. Due to the use of clean, we even have $P_0 \xrightarrow{v_0} P_{i_1} \xrightarrow{v_1} P_{i_2} \ldots$.

In the following example we use the same process as in Example 1 to provide an intuitive idea on how (weak) fairness of actions works and on its relationship with timing.

Example 2 Consider again $P = (R \parallel_\emptyset W) \parallel_{\{r,w\}} V$ and a run by P consisting of infinitely many r's. Such a computation is fair w.r.t. actions; in particular, it is fair for 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 fairness formally. After a time step, all activated actions become urgent and must be performed or get disabled before the next time step will be possible; this happens when r is performed, as we noted above:

$$P \xrightarrow{1} ((\mathsf{rec}\, x.\, \underline{r}.x) \parallel_\emptyset (\mathsf{rec}\, x.\, \underline{w}.x)) \parallel_{\{r,w\}} \mathsf{rec}\, x.\, (\underline{r}.x + \underline{w}.x) \xrightarrow{r} P$$

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 can indefinitely block another process trying to write to it (and vice versa for repeated writing). This kind of computations will be prevented by fairness of components as we will discuss in Example 4.

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

In this section, we concentrate on weak *fairness of components*. Not surprisingly, the PAFAS timed operational semantics does not support a characterisation of fair behaviour w.r.t. components. But we have found a suitable variation of PAFAS and its semantics which allows us to characterise Costa and Stirling's fairness of components again in terms of a simple filtering of system executions. Conceptually, we proceed analogously to Sect. 2.1, but a number of technical changes are needed to define the new semantics.

The new operational semantics of processes we have arrived at can again be understood as the behaviour of timed processes with upper time bounds. We assume that for each parallel component this upper time bound is 1; hence, a component will perform some action within time 1 provided it is continually enabled. In other words, when time 1 passes, an enabled component becomes urgent and, before the next time step, it must perform an action (or get disabled).

Also in this case, we assume that time elapses in a discrete way. Thus, an action-prefixed process $a.P$ can again either do action a and become process P (as usual in CCS) or can let one unit of time pass and become $\underline{a}.P$; $\underline{a}.P$ cannot let time pass, but can only do a to become P. Since we associate time bounds with components in the present section, we also mark 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$^{\mathsf{c}}$ henceforth.

Definition 9 (*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

following grammar:

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

where $P \in \tilde{\mathbb{P}}_1$, $x \in \mathscr{X}$, $\alpha \in \mathbb{A}_\tau$, 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 $\mathbb{P}_c$.

For studying fairness, we are interested in the initial processes, and these coincide in PAFAS and in PAFAS$^{\mathsf{C}}$; they are actually common CCS/TCSP-like processes. The additional terms of $\tilde{\mathbb{P}}_c$ turn up in evolutions of terms from $\tilde{\mathbb{P}}_1$ involving time steps, and here PAFAS and PAFAS$^{\mathsf{C}}$ differ.

We define function $\mathsf{A}(\_)$ on process terms, that returns the *activated* (or enabled) actions of a process term. Given a process Q, $\mathsf{A}(Q)$ again abbreviates $\mathsf{A}(Q, \emptyset)$ and $\mathsf{A}(Q, A)$ denotes the set of actions that process Q can perform when the environment prevents the actions in $A \subseteq \mathbb{A}$.

Definition 10 (*activated basic actions*) Let $Q \in \tilde{\mathbb{P}}_c$ and $A \subseteq \mathbb{A}$. The set $\mathsf{A}(Q, A)$ can be defined as in Definition 2 when rule Sum is replaced as follows:

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

2.2.1 The operational behaviour of PAFAS$^{\mathsf{C}}$ processes

The transitional semantics describing the functional behaviour of PAFAS$^{\mathsf{C}}$ processes indicates which basic actions they can perform. The operational semantics exploits two functions on process terms: $\mathsf{clean}(\_)$ and $\mathsf{unmark}(\_)$ with a similar meaning to those defined in the corresponding Section 2.1. Function $\mathsf{clean}(\_)$ *removes all inactive urgencies* in a process term $Q \in \tilde{\mathbb{P}}_c$. Indeed, when a process evolves, only whole components (and not single actions) may lose their urgency when their actions are no longer enabled due to changes of the context; 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 due to restrictions of the environment. Function $\mathsf{unmark}(\_)$ simply removes all urgencies (inactive or not) in a process term $Q \in \tilde{\mathbb{P}}_c$.

Definition 11 (*cleaning inactive urgencies*) Given a process term $Q \in \tilde{\mathbb{P}}_c$ 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 in Definition 4 where rules Sum and Par are replaced as follows:

$$\text{Sum: } \mathsf{clean}(P_1 + P_2, A) = P_1 + P_2 \quad \mathsf{clean}(P_1 \pm P_2, A) = \begin{cases} P_1 + P_2 & \text{if } \mathsf{A}(P_1) \cup \mathsf{A}(P_2) \subseteq A \\ P_1 \pm P_2 & \text{otherwise} \end{cases}$$

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

Definition 12 (*functional operational semantics*) The following SOS-rules define the transition relations $\overset{\alpha}{\longmapsto} \subseteq (\tilde{\mathbb{P}}_c \times \tilde{\mathbb{P}}_c)$ for $\alpha \in \mathbb{A}_\tau$, the *action transitions*; observe the use of $\longmapsto$ instead of $\rightarrow$.

$$\text{PREF}_{a1}\frac{}{\alpha.P \overset{\alpha}{\longmapsto} P} \qquad \text{PREF}_{a2}\frac{}{\underline{\alpha}.P \overset{\alpha}{\longmapsto} P}$$

$$\text{SUM}_{a1}\frac{P_1 \overset{\alpha}{\longmapsto} P_1'}{P_1 + P_2 \overset{\alpha}{\longmapsto} P_1'} \qquad \text{SUM}_{a2}\frac{P_1 \overset{\alpha}{\longmapsto} P_1'}{P_1 \mathbin{\underline{+}} P_2 \overset{\alpha}{\longmapsto} P_1'}$$

$$\text{PAR}_{a1}\frac{\alpha \notin A,\ Q_1 \overset{\alpha}{\longmapsto} Q_1'}{Q_1 \|_A Q_2 \overset{\alpha}{\longmapsto} \mathsf{clean}(Q_1' \|_A Q_2)} \qquad \text{PAR}_{a2}\frac{\alpha \in A,\ Q_1 \overset{\alpha}{\longmapsto} Q_1',\ Q_2 \overset{\alpha}{\longmapsto} Q_2'}{Q_1 \|_A Q_2 \overset{\alpha}{\longmapsto} \mathsf{clean}(Q_1' \|_A Q_2')}$$

$$\text{REL}_a\frac{Q \overset{\alpha}{\longmapsto} Q'}{Q[\Phi] \overset{\Phi(\alpha)}{\longmapsto} Q'[\Phi]} \qquad \text{REC}_a\frac{Q\{\mathsf{rec}\, x.\, \mathsf{unmark}(Q)/x\} \overset{\alpha}{\longmapsto} Q'}{\mathsf{rec}\, x.Q \overset{\alpha}{\longmapsto} Q'}$$

Additionally, there are symmetric rules for PAR_{a1}, SUM_{a1} and SUM_{a2} for actions of P_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 $P_0 \overset{\alpha_0}{\longmapsto} P_1 \overset{\alpha_1}{\longmapsto} \cdots$ of action transitions, possibly ending with a process P_n.

Observe the following for SUM_{a2}: due to our syntax, P_1 in $P_1 \mathbin{\underline{+}} P_2$ is an initial process, i.e. has no components marked as urgent, and the same applies to P_1'. Thus, $P_1 \mathbin{\underline{+}} P_2$ loses its urgency in a transition according to SUM_{a2}; this corresponds to our intuition, since this atomic component (i.e. without parallel subcomponents) performs an action, which it had to perform urgently, and can afterwards wait with any further activity for time 1.

2.2.2 The temporal behaviour of PAFAS^c process

In addition to the purely functional transitions, we also consider transitions corresponding to the passage of one unit of time. The function urgent we exploit marks the *enabled* parallel components of a process as urgent; such a component can be identified with a dynamic operator (an action or a choice), which gets underlined. This marking occurs when a time step is performed, because afterwards the marked components have to act in zero time – unless they are disabled. If such an urgent component acts, it should lose its urgency; and indeed, the marking vanishes with the dynamic operator. The next time step will only be possible, if no component is marked as urgent.

Definition 13 (*time step, timed execution sequence*) For $P \in \tilde{\mathbb{P}}_1$, we write $P \overset{1}{\longmapsto} Q$ when $Q = \mathsf{urgent}(P)$, where $\mathsf{urgent}(P)$ abbreviates $\mathsf{urgent}(P, \emptyset)$ and $\mathsf{urgent}(P, A)$ is defined as in Definition 7 but rule Sum is replaced as follows:

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

As in the corresponding Section 2.1.2, we define *timed execution sequences* to be infinite sequences of action transitions and time steps starting with some $P \overset{1}{\longmapsto} Q_0$ (again a maximal sequence of such transitions/steps starting is never finite) and the property *non-Zeno*.

Example 3 Again we provide an example for the use of the various definitions and for this we use process P already introduced in Example 1. According to the transitional rules in Definitions 12 and 13 we have:

$$\begin{aligned} P &\overset{1}{\longmapsto} (\underline{R} \parallel_{\emptyset} \underline{W}) \parallel_{\{r,w\}} \underline{V} = ((\mathsf{rec}\, x.\ \underline{r}.x) \parallel_{\emptyset} (\mathsf{rec}\, x.\ \underline{w}.x)) \parallel_{\{r,w\}} \mathsf{rec}\, x.\ (r.x + w.x) \\ &\overset{r}{\longmapsto} (R \parallel_{\emptyset} \underline{W}) \parallel_{\{r,w\}} V \\ &\overset{r}{\longmapsto} (R \parallel_{\emptyset} \underline{W}) \parallel_{\{r,w\}} V \end{aligned}$$

After one time unit, all the enabled parallel components become urgent in process P. In the second transition, $\underline{R}$ and $\underline{V}$ synchronize on action r. In the case of fairness of components, component $\underline{W}$ remains enabled and thus urgent, and it will indeed remain urgent until it synchronizes on the action w; cf. Definition 11, where the Par-case refers to activated actions of the context – and not to urgent ones as in Definition 4. We refer the reader to [3] for a detailed description of the various definitions.

2.2.3 Fairness of components and timing

As in the corresponding Section 2.1.3, 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 8 by replacing each action transition $\overset{\alpha}{\rightarrow}$ and time step $\overset{1}{\rightarrow}$ with its counterpart in the component-oriented timed operational semantics, i.e. $\overset{\alpha}{\longmapsto}$ and $\overset{1}{\longmapsto}$. To keep things short, we do not report here the formal definition.

Example 4 In the case of fairness of components, the infinite computation of r's in Example 2 is not fair intuitively because component W is enabled at every stage but it never performs w.

As already shown above, also in the component case this intuition corresponds to our formal definition in terms of timed runs, where components play the role of actions. Whenever P lets time pass evolving to $(\underline{R} \parallel_{\emptyset} \underline{W}) \parallel_{\{r,w\}} \underline{V}$, then a new time step is only possible when components R, W and V evolve by performing actions r and w. As a consequence, in any fair trace of P an infinite sequence of readings cannot indefinitely block the other process wishing to write the variable.

We conclude this section stating some properties of functions $\mathsf{urgent}(\_)$ and $\mathsf{unmark}(\_)$ which will be useful in the rest of the paper. A detailed proof of the statements can be found in [3].

Lemma 1 *Let* $Q \in \tilde{\mathbb{P}}_c$ *and* $P = \mathsf{unmark}(Q) \in \tilde{\mathbb{P}}_1$. *Then:*

1. $Q \overset{\alpha}{\longmapsto} Q'$ *implies* $P \overset{\alpha}{\longmapsto} P'$ *with* $P' = \mathsf{unmark}(Q')$;
2. $P \overset{\alpha}{\longmapsto} P'$ *implies* $Q \overset{\alpha}{\longmapsto} Q'$ *with* $P' = \mathsf{unmark}(Q')$.

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, but taken to be 1 here.

Informally, the b variables are "request" variables and the variable k is a "turn" variable. Variable b_i is *true* if $\mathtt{P}_i$ is requesting entry to its critical section; variable k is i if it is $\mathtt{P}_i$'s turn to enter its critical section. Only $\mathtt{P}_i$ writes the variable 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;
```

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 [15]—with some trivial changes due to the different language we are using (PAFAS instead of CCS). Each program variable is represented as a family of processes. Thus, 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 $\{b_1rt, b_1rf, b_1wf, b_1wt\}$ 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. In the following we denote with $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$ the set of boolean values and with $\mathbb{K} = \{1, 2\}$ the set of values that the variable k may take.

Definition 14 (*program variables*) Let $i \in \{1, 2\}$. We define the processes representing program variables as follows:

$$\mathsf{B}_i(\mathit{false}) = b_irf.\mathsf{B}_i(\mathit{false}) + (b_iwf.\mathsf{B}_i(\mathit{false}) + b_iwt.\mathsf{B}_i(\mathit{true}))$$
$$\mathsf{B}_i(\mathit{true}) = b_irt.\mathsf{B}_i(\mathit{true}) + (b_iwf.\mathsf{B}_i(\mathit{false}) + b_iwt.\mathsf{B}_i(\mathit{true}))$$

$$\mathsf{K}(1) = \mathit{kr1}.\mathsf{K}(1) + (\mathit{kw1}.\mathsf{K}(1) + \mathit{kw2}.\mathsf{K}(2))$$
$$\mathsf{K}(2) = \mathit{kr2}.\mathsf{K}(2) + (\mathit{kw1}.\mathsf{K}(1) + \mathit{kw2}.\mathsf{K}(2))$$

Let $B = \{b_irf, b_irt, b_iwf, b_iwt \mid i \in \{1, 2\}\} \cup \{\mathit{kr1}, \mathit{kr2}, \mathit{kw1}, \mathit{kw2}\}$ 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 $b_1, b_2 \in \mathbb{B}$ and $k \in \mathbb{K}$, we define

$$\mathsf{PV}(b_1, b_2, k) = (\mathsf{B}_1(b_1) \parallel \mathsf{B}_2(b_2)) \parallel \mathsf{K}(k),$$

where we use $\parallel$ as a shorthand for $\parallel_\emptyset$.

The next definition introduces the PAFAS version of Dekker's algorithm.

Definition 15 (*the algorithm*) The processes $\mathtt{P}_1$ and $\mathtt{P}_2$ are represented by the following PAFAS processes; the actions $\mathtt{req}_i$ and cs_i have been added to indicate the request to enter

and the execution of the critical section by the process $\mathtt{P}_i$.

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

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

The definition of Dekker in PAFAS closely follows the one in CCS given by Walker in [15]. Only a couple of changes have been implemented, and neither of them has an influence on the validity of the safety property. They all concern the definitions of P_1 (and P_2 resp.).

In [15], $P_1 = b_1wt.\mathtt{req}_1.P_{11}$, i.e. process $\mathtt{P}_1$ first sets its request variable to true and then performs the request. Walker regarded $\mathtt{req}_1$ as a *probe*, which just indicates that requesting has started. But there is a problem: in order to request access, $\mathtt{P}_1$ must first set its request variable , which is also used by the other process; therefore, the other process might block the request. This can really happen even under a reasonable fairness assumption as we will demonstrate below. Such a behaviour would give the wrong impression that the liveness property is satisfied—$\mathtt{P}_1$ just never requested; to avoid such situations, requesting must be completely under the control of the respective process, and to ensure this we have swapped the write and the request actions.

Furthermore, we have added a $\tau.P_1$-summand; without it, any of the two forms of weak fairness would force $\mathtt{P}_1$ to request. This is not desirable, since in reality $\mathtt{P}_1$ can decide to stay idle. In fact, with forced requesting the simple strategy of granting access to the critical section in turns would solve the mutex problem easily. In order to consider the real, much more difficult problem, one has to model the situation in which process P_1 decides to idle, and that we have done.

The third change is motivated by economy: for our approach, it suffices to have one action $\mathtt{cs}_1$ to denote performance of the critical section instead of two actions for entering and exiting as in [15].

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 16 (*live*) 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 $\gamma = P_0 \xrightarrow{1} Q_0 \xrightarrow{\lambda_0} Q_1 \xrightarrow{\lambda_1} \cdots$ ($\gamma = P_0 \longmapsto Q_0 \longmapsto Q_1 \longmapsto \cdots$) 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 $Q_{j+1} \xrightarrow{\lambda_{j+1}} Q_{j+2} \xrightarrow{\lambda_{j+2}} \cdots$ ($Q_{j+1} \overset{\lambda_{j+1}}{\longmapsto} Q_{j+2} \overset{\lambda_{j+2}}{\longmapsto} \cdots$ 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. For this purpose, 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. Sect. 2.1) which are not live. First of all, we need to describe how program variables and the processes $\mathtt{P}_1$ and $\mathtt{P}_2$ evolve by letting one time unit pass.

Definition 17 (*urgent program variables*) According to Definitions 14 and 7, *urgent program variables* can be defined as follows:

$$\underline{\mathsf{B}_i}(\mathit{false}) = \underline{b_i rf}.\mathsf{B}_i(\mathit{false}) + (\underline{b_i wf}.\mathsf{B}_i(\mathit{false}) + \underline{b_i wt}.\mathsf{B}_i(\mathit{true}))$$
$$\underline{\mathsf{B}_i}(\mathit{true}) = \underline{b_i rt}.\mathsf{B}_i(\mathit{true}) + (\underline{b_i wf}.\mathsf{B}_i(\mathit{false}) + \underline{b_i wt}.\mathsf{B}_i(\mathit{true}))$$

$$\underline{\mathsf{K}}(1) = \underline{kr1}.\mathsf{K}(1) + (\underline{kw1}.\mathsf{K}(1) + \underline{kw2}.\mathsf{K}(2))$$
$$\underline{\mathsf{K}}(2) = \underline{kr2}.\mathsf{K}(2) + (\underline{kw1}.\mathsf{K}(1) + \underline{kw2}.\mathsf{K}(2))$$

We further introduce $\underline{\mathbb{B}} = \{\underline{\mathit{false}}, \underline{\mathit{true}}\}$ and $\underline{\mathbb{K}} = \{\underline{1}, \underline{2}\}$. Then, given $b'_1, b'_2 \in \mathbb{B} \cup \underline{\mathbb{B}}$ and $k' \in \mathbb{K} \cup \underline{\mathbb{K}}$, we define $\mathsf{PV}(b'_1, \overline{b'_2, k'}) = ((B_1 \parallel B_2) \parallel K)$, 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 $\mathsf{PV}(\underline{\mathit{true}}, \mathit{false}, \underline{2}) = (\underline{\mathsf{B}_1}(\mathit{true}) \parallel \mathsf{B}_2(\mathit{false})) \parallel \underline{\mathsf{K}}(2)$.

Similarly, the urgent versions of processes $\mathtt{P}_1$ and $\mathtt{P}_2$ can be defined as follows:

Definition 18 (*urgent processes*)

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

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 $\underline{\mathit{Dekker}}$, where:

$$\underline{\mathit{Dekker}} = ((\underline{P_1} \parallel \underline{P_2}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B]$$

Our first example shows how an infinite τ-loop can result in the starvation of both processes.

Example 5 Let us consider the following timed computation from *Dekker*:

$$
\begin{aligned}
\mathit{Dekker} \xrightarrow{1} \underline{\mathit{Dekker}} = ((\underline{P_1} \parallel \underline{P_2}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] \quad & \xrightarrow{\mathtt{req}_1} \xrightarrow{\mathtt{req}_2} \\
((b_1 wt.P_{11} \parallel b_2 wt.P_{21}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((P_{11} \parallel b_2 wt.P_{21}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{false}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((P_{11} \parallel P_{21}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((P_{11} \parallel P_{22}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
\\
P_0 = ((P_{11} \parallel b_2 wf.P_{23}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{1} \\
Q_0 = ((\underline{P_{11}} \parallel \underline{b_2 wf}.P_{23}) \parallel_B \mathsf{PV}(\mathit{true}, \underline{\mathit{true}}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((P_{12} \parallel b_2 wf.P_{23}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
P_0 = ((P_{11} \parallel b_2 wf.P_{23}) \parallel_B \mathsf{PV}(\mathit{true}, \mathit{true}, 1))[\Phi_B] &
\end{aligned}
$$

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. More 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. We have already indicated in Example 1 how fairness of actions allows computations along which repeated accesses to a variable can block another access. On the other hand, $\mathtt{P}_1$ cannot enter its critical section and, hence, cannot proceed until the value of b_2 is true. As a consequence, both processes are stuck; this undesirable behaviour would not be detected with the liveness definition of [15], since no process enters its critical section.

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.

Example 6 Now, consider the following timed computation:

$$
\begin{aligned}
\mathit{Dekker} \xrightarrow{1} \underline{\mathit{Dekker}} = ((\underline{P_1} \parallel \underline{P_2}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] \quad & \xrightarrow{\mathtt{req}_1} \xrightarrow{\mathtt{req}_2} \\
P_0 = ((b_1 wt.P_{11} \parallel b_2 wt.P_{21}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] \quad & \xrightarrow{1} \\
((\underline{b_1 wt}.P_{11} \parallel \underline{b_2 wt}.P_{21}) \parallel_B \mathsf{PV}(\underline{\mathit{false}}, \underline{\mathit{false}}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((\underline{b_1 wt}.P_{11} \parallel P_{21}) \parallel_B \mathsf{PV}(\underline{\mathit{false}}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((b_1 wt.P_{11} \parallel P_{24}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{\mathtt{cs}_2} \\
((b_1 wt.P_{11} \parallel kw1.b_2 wf.P) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((b_1 wt.P_{11} \parallel b_2 wf.P_2) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{true}, 1))[\Phi_B] \quad & \xrightarrow{\tau} \\
((b_1 wt.P_{11} \parallel P_2) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] \quad & \xrightarrow{\mathtt{req}_2} \\
((b_1 wt.P_{11} \parallel b_2 wt.P_{21}) \parallel_B \mathsf{PV}(\mathit{false}, \mathit{false}, 1))[\Phi_B] = P_0 &
\end{aligned}
$$

Again, the trace performed in

$$Dekker \xrightarrow{\mathtt{req}_1\,\mathtt{req}_2} P_0 \xrightarrow{\tau^2\,\mathtt{cs}_2\,\tau\tau\,\mathtt{req}_2} P_0 \ldots$$

is fair but violates liveness since $\mathtt{P}_1$ never enters its critical section. Here, $\mathtt{P}_2$ repeatedly executes its critical section, again preventing $\mathtt{P}_1$ to set its request variable b_1 to *true*. As a consequence, $\mathtt{P}_1$ cannot enter its critical section even though the value of turn variable k is 1. In other words, writing a value to a variable (b_1wt) is blocked by repeated reading of the variable (b_1rt); accordingly, $\mathtt{P}_1$ could be prevented from requesting in the modelling of Walker.

5 Fairness of components and liveness

This section is the core of the paper. It proves that any fair trace of *Dekker* according to fairness of components satisfies the liveness property, i.e. that any non-Zeno timed execution sequence of *Dekker* in PAFAS$^{\mathsf{c}}$ is live in the sense of Definition 16. We will present three ideas that help to reduce the number of states we have to deal with.

5.1 Permanently lazy components

The state space of a process in PAFAS$^{\mathsf{c}}$ is considerably larger than in an untimed process algebra because process components switch from lazy to urgent. In principle, for a process with 5 components like *Dekker*, this could make the state space 32-fold in size. We can achieve a considerable reduction, if we prevent this by declaring some components as permanently lazy, as we describe it in this subsection. As an application, we will regard the three program variables as one component of *Dekker* for technical reasons, and declare it as permanently lazy; this results in a process denoted by $Dekker[\mathsf{PV}]$.

Technically, this declaration means that a non-Zeno timed execution sequence of the original process can be simulated by one of the new process, but not vice versa. This way, 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 $Dekker[\mathsf{PV}]$ are live.

Although this is only a sufficient condition, it should better be satisfied in our example since there is a good intuitive reason behind it. 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 $\mathtt{P}_2$. As it turns out, assuming this kind of fairness is indeed enough.

We now extend PAFAS$^{\mathsf{c}}$ with a new operator, which intuitively speaking can only be applied to a top-level component.

Definition 19 (*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:

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

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

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

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.

Next, we define the operational semantics for processes with one permanently lazy top-level component. As in PAFAS$^{\mathsf{c}}$, the respective SOS-rules exploit a function clean(_) on process terms which, as in the previous sections, removes all inactive urgencies, and we also use a function unmark(_) which removes all urgencies (inactive or not). Both these functions can be defined by induction on the process structure as follows:

Definition 20 (*cleaning urgencies*) Let R be a $\tilde{\mathbb{P}}_\ell$ term. Then $\mathsf{clean}(R)$ and $\mathsf{unmark}(R)$ are defined by induction on R as follows:

Par: $\mathsf{clean}(Q \parallel_A [P]) = \mathsf{clean}(Q, A_1) \parallel_A [P] \quad$ where $A_1 = (\mathsf{A}(Q) \backslash \mathsf{A}(P)) \cap A$
Rel: $\mathsf{clean}(R[\Phi]) = \mathsf{clean}(R)[\Phi]$

Par: $\mathsf{unmark}(Q \parallel_A [P]) = \mathsf{unmark}(Q) \parallel_A [P]$
Rel: $\mathsf{unmark}(R[\Phi]) = \mathsf{unmark}(R)[\Phi]$

The functional behaviour of $\tilde{\mathbb{P}}_\ell$ terms can be defined as follows:

Definition 21 (*Functional operational semantics*) The following SOS-rules define the transition relations $\overset{\alpha}{\longmapsto} \subseteq (\tilde{\mathbb{P}}_\ell \times \tilde{\mathbb{P}}_\ell)$ for $\alpha \in \mathbb{A}_\tau$, the *action transitions*.

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

$$\textsc{LazySynch}\frac{\alpha \in A,\ Q \overset{\alpha}{\longmapsto} Q',\ P \overset{\alpha}{\longmapsto} P'}{Q \parallel_A [P] \overset{\alpha}{\longmapsto} \mathsf{clean}(Q' \parallel_A [P'])} \qquad \textsc{LazyRel}_{a}\frac{R \overset{\alpha}{\longmapsto} R'}{R[\Phi] \overset{\Phi(\alpha)}{\longmapsto} R'[\Phi]}$$

Definition 22 (*time step*) For $S \in \tilde{\mathbb{P}}_{\ell 1}$, we write that $S \overset{1}{\longmapsto} R$ when $R = \mathsf{urgent}(S)$ where function $\mathsf{urgent}(S)$ is defined as follows:

Par: $\mathsf{urgent}(P_1 \parallel_A [P_2]) = \mathsf{urgent}(P_1, A_1) \parallel_A [P_2] \quad$ where $A_1 = (\mathsf{A}(P_1) \backslash \mathsf{A}(P_2)) \cap A$

Rel: $\mathsf{urgent}(S[\Phi]) = \mathsf{urgent}(S)[\Phi]$

Definition 23 Let $Q \in \tilde{\mathbb{P}}$ and $R \in \tilde{\mathbb{P}}_\ell$. We write that $Q \prec R$ 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 $Q_1 \prec R_1$.

Let us denote by $Dekker[\mathsf{PV}]$ the process $((P_1 \parallel_B P_2) \parallel [\mathsf{PV}(false, false, 1)])[\Phi_B]$, which is *Dekker* with permanently lazy process variables. We prove that any non-Zeno timed execution sequence of *Dekker* can be simulated by a corresponding non-Zeno timed execution sequence of $Dekker[\mathsf{PV}]$ (see Proposition 3). This allows us to consider, for our calculation, $Dekker[\mathsf{PV}]$ (instead of *Dekker*).

Lemma 2 *Let* $Q \in \tilde{\mathbb{P}}$ *and* $R, R' \in \tilde{\mathbb{P}}_\ell$. *Then*:

1. $Q \prec R$ *and* $Q \in \tilde{\mathbb{P}}_1$ *imply* $R \in \tilde{\mathbb{P}}_{\ell 1}$;
2. $Q \prec R$ *and* $Q \prec R'$ *imply* $R = R'$.

Proposition 1 *Let* $Q \in \tilde{\mathbb{P}}$ *and* $R \in \tilde{\mathbb{P}}_\ell$ *with* $Q \prec R$. *Then*:

1. $Q \stackrel{\alpha}{\longmapsto} Q'$ *implies* $R \stackrel{\alpha}{\longmapsto} R'$ *and* $Q' \prec R'$;
2. $R \stackrel{\alpha}{\longmapsto} R'$ *implies* $Q \stackrel{\alpha}{\longmapsto} Q'$ *and* $Q' \prec R'$.

Proof We only prove Item 1 (Item 2 can be proved similarly). By Definition 23 we have to consider two possible cases:

Par: $Q = Q_1 \parallel_A Q_2$ and $R = Q_1 \parallel_A [P_2]$ with $P_2 = \mathsf{unmark}(Q_2)$. Assume that $Q \stackrel{\alpha}{\longmapsto} Q'$ and consider the following possible cases:

- $\alpha \notin A$, $Q_1 \stackrel{\alpha}{\longmapsto} Q_1'$ and $Q' = \mathsf{clean}(Q_1' \parallel_A Q_2) = \mathsf{clean}(Q_1', A_1) \parallel_A \mathsf{clean}(Q_2, A_2)$ with $A_1 = (\mathsf{A}(Q_1')\backslash\mathsf{A}(Q_2)) \cap A = (\mathsf{A}(Q_1')\backslash\mathsf{A}(P_2)) \cap A$ (since $\mathsf{A}(Q_2) = \mathsf{A}(\mathsf{unmark}(Q_2)) = \mathsf{A}(P_2)$) and $A_2 = (\mathsf{A}(Q_2)\backslash\mathsf{A}(Q_1')) \cap A$. On the other hand $\alpha \notin A$ and $Q_1 \stackrel{\alpha}{\longmapsto} Q_1'$ imply, by rule LazyPar$_{a1}$, $R \stackrel{\alpha}{\longmapsto} \mathsf{clean}(Q_1' \parallel_A [P_2]) = \mathsf{clean}(Q_1', A_1) \parallel_A [P_2] = R'$. Finally, $\mathsf{unmark}(\mathsf{clean}(Q_2, A_2)) = \mathsf{unmark}(Q_2) = P_2$ implies $Q' \prec R'$.
- $\alpha \notin A$, $Q_2 \stackrel{\alpha}{\longmapsto} Q_2'$ and $Q' = \mathsf{clean}(Q_1 \parallel_A Q_2') = \mathsf{clean}(Q_1, A_1) \parallel_A \mathsf{clean}(Q_2', A_2)$ with $A_1 = (\mathsf{A}(Q_1)\backslash\mathsf{A}(Q_2')) \cap A$ and $A_2 = (\mathsf{A}(Q_2')\backslash\mathsf{A}(Q_1)) \cap A$. Moreover, by Lemma 1, $Q_2 \stackrel{\alpha}{\longmapsto} Q_2'$ implies $P_2 = \mathsf{unmark}(Q_2) \stackrel{\alpha}{\longmapsto} \mathsf{unmark}(Q_2') = P_2'$. Thus, $R \stackrel{\alpha}{\longmapsto} \mathsf{clean}(Q_1 \parallel_A [P_2']) = \mathsf{clean}(Q_1, A_1) \parallel_A [P_2'] = R'$ (since $(\mathsf{A}(Q_1)\backslash\mathsf{A}(P_2')) \cap A = (\mathsf{A}(Q_1)\backslash\mathsf{A}(Q_2')) \cap A = A_1$). Finally, $\mathsf{unmark}(\mathsf{clean}(Q_2', A_2)) = \mathsf{unmark}(Q_2') = P_2'$ implies $Q' \prec R'$.
- $\alpha \in A$, $Q_1 \stackrel{\alpha}{\longmapsto} Q_1'$, $Q_2 \stackrel{\alpha}{\longmapsto} Q_2'$ and $Q' = \mathsf{clean}(Q_1' \parallel_A Q_2')$. A mix of the proofs of the above two cases.

Rel: $Q = Q_1[\Phi]$ and $R = R_1[\Phi]$ with $Q_1 \prec R_1$. In this case $Q \stackrel{\alpha}{\longmapsto} Q'$ if there exists $\beta \in \Phi^{-1}(\alpha)$ such that $Q_1 \stackrel{\beta}{\longmapsto} Q_1'$ and $Q' = Q_1'[\Phi]$. By induction hypothesis we have that $R_1 \stackrel{\beta}{\longmapsto} R_1'$ with $Q_1' \prec R_1'$. Thus, $R \stackrel{\alpha}{\longmapsto} R_1'[\Phi] = R'$ and $Q' \prec R'$.

Proposition 2 *Let* $P \in \tilde{\mathbb{P}}_1$ *and* $S \in \tilde{\mathbb{P}}_{\ell 1}$ *with* $P \prec S$. *Then* $\mathsf{urgent}(P) \prec \mathsf{urgent}(S)$.

Proof Again we have the following two possible cases:

Par: $P = P_1 \parallel_A P_2$ and $S = P_1 \parallel_A [P_2]$. $\mathsf{urgent}(P_1 \parallel_A P_2) = \mathsf{urgent}(P_1, A_1) \parallel_A \mathsf{urgent}(P_2, A_2) = Q_1 \parallel_B Q_2$ where $A_1 = (\mathsf{A}(P_1)\backslash\mathsf{A}(P_2)) \cap A$ and $A_2 = (\mathsf{A}(P_2)\backslash\mathsf{A}(P_1)) \cap A$. On the other hand, $\mathsf{urgent}(P_1 \parallel_A [P_2]) = \mathsf{urgent}(P_1, A_1) \parallel_A [P_2] = Q_1 \parallel_A [P_2] = R$. Finally, $\mathsf{unmark}(Q_2) = \mathsf{unmark}(\mathsf{urgent}(P_2, A_2)) = P_2$ implies $\mathsf{urgent}(P) \prec \mathsf{urgent}(S)$.

Rel: $P = P_1[\Phi]$ and $S = S_1[\Phi]$ with $P_1 \prec S_1$. By induction hypothesis, we have also $\mathsf{urgent}(P) = \mathsf{urgent}(P_1)[\Phi] \prec \mathsf{urgent}(S_1)[\Phi] = \mathsf{urgent}(S)$.

By iterative applications of Propositions 1 and 2 we can prove the following statement:

Proposition 3 *Let $P \in \tilde{\mathbb{P}}_1$, $S \in \tilde{\mathbb{P}}_{\ell 1}$ with $P \prec S$ and $v \in (\mathbb{A}_\tau)^*$. Then $P \xmapsto{1} Q \xmapsto{v} P' \in \tilde{\mathbb{P}}_1$ implies $S \xmapsto{1} R \xmapsto{v} S'$ with $P' \prec S'$ (and hence S can simulate each non-Zeno timed execution sequence of P).*

5.2 F-Steps

We can group the transitions of a non-Zeno timed execution sequence into infinitely many steps of the form $S \xmapsto{1} R \xmapsto{v} S'$, 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 $R \xmapsto{v} S'$ 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 many such f-steps.

Definition 24 (*f-executions*) A transition sequence $S \xmapsto{1} R \xmapsto{v} S'$ 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 $R \xmapsto{v} S'$ (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:

$$\gamma = S_0 \xmapsto{1} R_0 \xmapsto{v_0} S_1 \xmapsto{1} R_1 \xmapsto{v_1} S_2 \ldots$$

We call the processes $S_1, S_2, \ldots$ *fair-reachable* from S_0.

According to this definition, f-executions are special non-Zeno timed execution sequences. To show that checking them for liveness suffices, we need the following lemmas.

Lemma 3 *Let $P_0, P_1, P_2 \in \mathbb{P}_1$, v_0 and $v_1 \in (\mathbb{A}_\tau)^*$. Then: $P_0 \xmapsto{v_0} P_1 \xmapsto{1} Q_1 \xmapsto{v_1} P_2$ implies $P_0 \xmapsto{1} Q_0 \xmapsto{v_0 v_1} P_2$.*

Proof This can be proven by collecting some results stated in [3].

Lemma 4 *Let $S_0, S_1, S_2 \in \mathbb{P}_{\ell 1}$, v_0 and $v_1 \in (\mathbb{A}_\tau)^*$. Then: $S_0 \xmapsto{v_0} S_1 \xmapsto{1} R_1 \xmapsto{v_1} S_2$ implies $S_0 \xmapsto{1} R_0 \xmapsto{v_0 v_1} S_2$.*

Proof Let $S_0 \in \mathbb{P}_{\ell 1}$ and let P_0 be the $\mathbb{P}_1$-process obtained from S_0 by removing laziness from its top-level permanent lazy component. By Definition 23, it is $P_0 \prec S_0$. Now, assume that $S_0 \xmapsto{v_0} S_1 \xmapsto{1} R_1 \xmapsto{v_1} S_2$. Then $P_0 \prec S_0$, Propositions 1-2 and 3 imply $P_0 \xmapsto{v_0} P_1 \xmapsto{1} Q_1 \xmapsto{v_1} P_2$ with $P_i \prec S_i$, for $i \in \{1, 2\}$. By Lemma 3 and (iterative applications of) Proposition 1-1, we also have that $P_0 \xmapsto{1} Q_0 \xmapsto{v_0 v_1} P_2$ and $S_0 \xmapsto{1} R_0 \xmapsto{v_0 v_1} S_2'$ with $P_2 \prec S_2'$. Finally, $P_2 \prec S_2'$ and $P_2 \prec S_2$ imply $S_2' = S_2$ (see Lemma 2-2).

Lemma 5 *Let $S_0, S_1, S_2 \in \mathbb{P}_{\ell 1}$, $v_1 \in (\mathbb{A}_\tau)^*$ and $\delta_0, \delta_1 \in (\mathbb{A}_\tau \cup \{1\})^*$. If $S_0 \xmapsto{\delta_0} S_1 \xmapsto{v_1} S_2 \xmapsto{1} R_2 \xmapsto{\delta_1}$ is a non-Zeno timed execution sequence, then $S_0 \xmapsto{\delta_0} S_1 \xmapsto{1} R_1 \xmapsto{v_1 \delta_1}$ is also a non-Zeno timed execution sequence.*

Proof Directly from Lemma 4, since in δ_1 we have infinitely many time-steps and, hence, "along δ_1" there will be an initial process playing the rôle of S_2 in Lemma 4.

Proposition 4 *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.*

Proof We use Lemma 5 repeatedly to move each time step to the first possible position. Observe that γ either deadlocks (ends with some process S_k performing $(\stackrel{1}{\longmapsto})^\omega$) or always performs one or more actions between two consecutive time-steps, so the limit contains all actions.

By definition of fair-traces, Propositions 3 and 4, in order to check the liveness property of Dekker's algorithm for all fair traces, we only have to check that all f-executions of *Dekker*[PV] are live. The initial processes we have to consider for this check must all be fair-reachable from *Dekker*[PV].

5.3 Symmetry of fair-reachable processes

Half of the processes which are fair-reachable from *Dekker*[PV] are denoted by $D_1, \ldots, D_{47}$. Their formal definition is provided in Tables 1 and 2 while a detailed description of the way in which they have been obtained can be found the technical report in [5].[3] 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. Namely, for each $y \in [1, 47]$, $S_y = \mathscr{S}(D_y)$, where the function $\mathscr{S}(\_)$ on processes is given below. Moreover, $\mathscr{S}(S_y) = D_y$ for any y.

Definition 25 (*symmetric processes*) Let $P_1, P_{11}, \ldots P_{14}, P_2, P_{21}, \ldots P_{24}$ be processes as given in Definition 15. Let moreover $x \in [1, 4]$ and $\{i, j\} = \{1, 2\}$. Then:

$$\begin{array}{ll} \mathscr{S}(P_i) = P_j & \mathscr{S}(P_{ix}) = P_{jx} \\ \mathscr{S}(b_i wt.P_{i1}) = b_j wt.P_{j1} & \mathscr{S}(b_i wf.P_{i3}) = b_j wf.P_{j3} \\ \mathscr{S}(kwj.b_i wf.P_i) = kwi.b_j wf.P_j & \mathscr{S}(b_i wf.P_i) = b_j wf.P_j \end{array}$$

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*[PV]. We can define the symmetric process of S as follows:

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

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

[3] There are 192 non-initial processes reached during the execution of f-steps from any $D_y \in \mathtt{D}$; they are listed in [5].

Table 1 Fair-reachable processes

$D_1 = ((b_1wt.P_{11} \parallel b_2wt.P_{21}) \parallel_B [\mathsf{PV}(false, false, 1)])[\Phi_B]$
$D_2 = ((b_1wt.P_{11} \parallel P_2) \parallel_B [\mathsf{PV}(false, false, 1)])[\Phi_B]$
$D_3 = ((P_1 \parallel b_2wt.P_{21}) \parallel_B [\mathsf{PV}(false, false, 1)])[\Phi_B]$
$D_4 = ((P_{11} \parallel b_2wt.P_{21}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]$
$D_5 = ((P_{11} \parallel P_2) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]$
$D_6 = ((b_1wt.P_{11} \parallel P_{21}) \parallel_B [\mathsf{PV}(false, true, 1)])[\Phi_B]$
$D_7 = ((P_1 \parallel P_{21}) \parallel_B [\mathsf{PV}(false, true, 1)])[\Phi_B]$
$D_8 = ((P_{14} \parallel b_2wt.P_{21}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]$
$D_9 = ((P_{14} \parallel P_2) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]$
$D_{10} = ((b_1wt.P_{11} \parallel P_{24}) \parallel_B [\mathsf{PV}(false, true, 1)])[\Phi_B]$
$D_{11} = ((P_1 \parallel P_{24}) \parallel_B [\mathsf{PV}(false, true, 1)])[\Phi_B]$
$D_{12} = ((kw2.b_1wf.P_1 \parallel b_2wt.P_{21}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]$
$D_{13} = ((kw2.b_1wf.P_1 \parallel P_2) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]$
$D_{14} = ((b_1wt.P_{11} \parallel kw1.b_2wf.P_2) \parallel_B [\mathsf{PV}(false, true, 1)])[\Phi_B]$
$D_{15} = ((P_1 \parallel kw1.b_2wf.P_2) \parallel_B [\mathsf{PV}(false, true, 1)])[\Phi_B]$
$D_{16} = ((b_1wf.P_1 \parallel b_2wt.P_{21}) \parallel_B [\mathsf{PV}(true, false, 2)])[\Phi_B]$
$D_{17} = ((b_1wf.P_1 \parallel P_2) \parallel_B [\mathsf{PV}(true, false, 2)])[\Phi_B]$
$D_{18} = ((P_{11} \parallel P_{21}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]$
$D_{19} = ((P_{14} \parallel P_{21}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]$
$D_{20} = ((P_{11} \parallel P_{24}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]$
$D_{21} = ((kw2.b_1wf.P_1 \parallel P_{21}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]$
$D_{22} = ((P_{11} \parallel kw1.b_2wf.P_2) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]$
$D_{23} = ((b_1wf.P_1 \parallel P_{21}) \parallel_B [\mathsf{PV}(true, true, 2)])[\Phi_B]$

Definition 26 (*symmetric sequences of actions*) Let $v \in (\mathbb{A}_d \cup \{\tau\})^*$. Then, the string $\mathscr{S}(v)$ can be defined, by induction on the length of v, as follows:

$$\mathscr{S}(v) = \begin{cases} \varepsilon & \text{if } v = \varepsilon \\ \tau\ \mathscr{S}(v') & \text{if } v = \tau\ v' \\ \mathtt{req}_j\ \mathscr{S}(v') & \text{if } v = \mathtt{req}_i\ v' \quad \text{and} \quad j = (i \mod 2) + 1 \\ \mathsf{cs}_j\ \mathscr{S}(v') & \text{if } v = \mathsf{cs}_i\ v' \quad \text{and} \quad j = (i \mod 2) + 1 \end{cases}$$

The following proposition simply states that symmetric processes have symmetric behaviors, i.e. they perform symmetric f-steps and then evolve into processes which are again symmetric.

Proposition 5 *Let $S \approx S'$ and $v \in (\mathbb{A}_d \cup \{\tau\})^*$. Then: $S' \xrightarrow{1} R' \xrightarrow{v} S'_0 \in \tilde{\mathbb{P}}_{\ell 1}$ implies $S \xrightarrow{1} R \xrightarrow{\mathscr{S}(v)} S_0 \in \tilde{\mathbb{P}}_{\ell 1}$ 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 states the main result of this section, i.e. all processes which are fair-reachable from $Dekker[\mathsf{PV}]$ belong to the set $\mathtt{D} \cup \mathtt{S}$.

Proposition 6 *Let $S \in \mathtt{D} \cup \mathtt{S}$ and $v \in (\mathbb{A}_d \cup \{\tau\})^*$. $S \xrightarrow{1} R \xrightarrow{v} S' \in \tilde{\mathbb{P}}_{\ell 1}$ implies $S' \in \mathtt{D} \cup \mathtt{S}$.*

Table 2 Fair-reachable processes

$$
\begin{aligned}
D_{24} &= ((P_{12} \parallel P_{21}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{25} &= ((P_{12} \parallel P_{22}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{26} &= ((P_{12} \parallel b_2 wf.P_{23}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{27} &= ((P_{14} \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]\\
D_{28} &= ((P_{11} \parallel P_{22}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{29} &= ((kw2.b_1 wf.P_1 \parallel P_{22}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{30} &= ((kw2.b_1 wf.P_1 \parallel b_2 wf.P_{23}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{31} &= ((kw2.b_1 wf.P_1 \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]\\
D_{32} &= ((P_{12} \parallel kw1.b_2 wf.P_1) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{33} &= ((b_1 wf.P_1 \parallel P_{22}) \parallel_B [\mathsf{PV}(true, true, 2)])[\Phi_B]\\
D_{34} &= ((b_1 wf.P_1 \parallel b_2 wf.P_{23}) \parallel_B [\mathsf{PV}(true, true, 2)])[\Phi_B]\\
D_{35} &= ((b_1 wf.P_1 \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 2)])[\Phi_B]\\
D_{36} &= ((P_1 \parallel P_{22}) \parallel_B [\mathsf{PV}(false, true, 2)])[\Phi_B]\\
D_{37} &= ((P_{11} \parallel b_2 wf.P_{23}) \parallel_B [\mathsf{PV}(true, true, 1)])[\Phi_B]\\
D_{38} &= ((P_{11} \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]\\
D_{39} &= ((P_{12} \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B]\\
D_{40} &= ((P_1 \parallel P_{23}) \parallel_B [\mathsf{PV}(false, false, 2)])[\Phi_B]\\
D_{41} &= ((b_1 wt.P_{11} \parallel P_{23}) \parallel_B [\mathsf{PV}(false, false, 2)])[\Phi_B]\\
D_{42} &= ((P_{11} \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 2)])[\Phi_B]\\
D_{43} &= ((P_{12} \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 2)])[\Phi_B]\\
D_{44} &= ((b_1 wf.P_{13} \parallel P_{23}) \parallel_B [\mathsf{PV}(true, false, 2)])[\Phi_B]\\
D_{45} &= ((P_{13} \parallel P_{23}) \parallel_B [\mathsf{PV}(false, false, 2)])[\Phi_B]\\
D_{46} &= (b_1 wf.P_{13} \parallel b_2 wt.P_{21}) \parallel_B [\mathsf{PV}(true, false, 2)])[\Phi_B]\\
D_{47} &= (P_{13} \parallel b_2 wt.P_{21}) \parallel_B [\mathsf{PV}(false, false, 2)])[\Phi_B]
\end{aligned}
$$

Proof The proof of the case in which $S \in \mathtt{D}$ can be found in [5] (see Lemma A.1). If $S \in \mathtt{S}$, the statement follows directly from Proposition 5.

5.4 Progressing processes

We can distinguish terms in $\mathtt{D} \cup \mathtt{S}$ depending on how many processes are waiting to perform their critical section or, in other words, depending on how many actions cs_i are still pending. We say that the action 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 $Dekker[\mathsf{PV}] \overset{v\ \mathtt{req}_i\ w}{\longmapsto} S$ and $\mathsf{cs}_i \notin w$. Each process may have at most two pending actions and so we are interested in the set $\mathtt{R}_1$ ($\mathtt{R}_2$) of fair-reachable states with only cs_1 (cs_2, respectively) pending and the set $\mathtt{R}_{1,2}$ of fair-reachable states with both cs_1 and 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. As an example, consider $D_{14} = ((b_1 wt.P_{11} \parallel kw1.b_2 wf.P_{21}) \parallel_B [\mathsf{PV}(false, true, 1)])[\Phi_B]$: $\mathtt{P}_1$ has requested to perform its critical section but the corresponding action cs_1 has not been performed yet; $\mathtt{P}_2$ is exiting its critical section and both actions $\mathtt{req}_2$ and cs_2 have already been executed. Thus, $D_{14} \in \mathtt{R}_1$. Similarly, it is $D_{12} = ((kw2.b_1 wf.P_1 \parallel b_2 wt.P_{21}) \parallel_B [\mathsf{PV}(true, false, 1)])[\Phi_B] \in \mathtt{R}_2$

while $D_1 = ((b_1wt.P_{11} \parallel b_2wt.P_{21}) \parallel_B [\mathsf{PV}(\mathit{false}, \mathit{false}, 1)])[\Phi_B] \in \mathtt{R}_{1,2}$. In detail, we have:

$$\begin{aligned}
\mathtt{D}_1 &= \mathtt{D} \cap \mathtt{R}_1 = \{D_2, D_5, D_9, D_{14}, D_{22}, D_{32}\} \\
\mathtt{D}_2 &= \mathtt{D} \cap \mathtt{R}_2 = \{D_3, D_7, D_{11}, D_{12}, D_{16}, D_{21}, D_{23}, D_{29}, D_{30}, D_{31}, D_{33}, \ldots, D_{36}, D_{40}\} \\
\mathtt{D}_{1,2} &= \mathtt{D} \cap \mathtt{R}_{1,2} \\
&= \{D_1, D_4, D_6, D_8, D_{10}, D_{18}, D_{19}, D_{20}, D_{24}, \ldots, D_{28}, D_{37}, D_{38}, D_{39}, D_{41}, \ldots, D_{47}\}
\end{aligned}$$

Processes D_0, D_{13}, D_{15} and D_{17} have no pending sections. Furthermore, since $S \in \mathtt{R}_1$, $S \in \mathtt{R}_2$ and $S \in \mathtt{R}_{1,2}$ imply $\mathscr{S}(S) \in \mathtt{R}_2$, $\mathscr{S}(S) \in \mathtt{R}_1$ and $\mathscr{S}(S) \in \mathtt{R}_{1,2}$, respectively, we also have: $\mathtt{S}_1 = \mathtt{S} \cap \mathtt{R}_1 = \mathscr{S}(\mathtt{D}_2)$, $\mathtt{S}_2 = \mathtt{S} \cap \mathtt{R}_2 = \mathscr{S}(\mathtt{D}_1)$ and $\mathtt{S}_{1,2} = \mathtt{S} \cap \mathtt{R}_{1,2} = \mathscr{S}(\mathtt{D}_{1,2})$. S_0, S_{13}, S_{15} and S_{17} have no pending actions.

Definition 27 (*progressing processes*) Let $S \in \mathtt{D} \cup \mathtt{S}$ and $v = \alpha_1 \ldots \alpha_n \in (\mathbb{A}_d \cup \{\tau\})^*$:

- We say that the string v contains the action $\mathtt{cs}_i$ ($i \in \{1, 2\}$), and write that $\mathtt{cs}_i \in v$ if $\alpha_j = \mathtt{cs}_i$ for some $j \in [1, n]$, and $\mathtt{cs}_1, \mathtt{cs}_2 \in v$ if both $\mathtt{cs}_1 \in v$ and $\mathtt{cs}_2 \in v$.
- We say that S implies the execution of the action $\mathtt{cs}_i$, written $S \rhd \mathtt{cs}_i$, if each f-execution from S contains the action $\mathtt{cs}_i$. We write $S \rhd \mathtt{cs}_1, \mathtt{cs}_2$ to denote that both $S \rhd \mathtt{cs}_1$ and $S \rhd \mathtt{cs}_2$. Finally, given $A \subseteq \mathtt{D} \cup \mathtt{S}$ we write $A \rhd \mathtt{cs}_i$ ($A \rhd \mathtt{cs}_1, \mathtt{cs}_2$) to denote that $S \rhd \mathtt{cs}_i$ ($S \rhd \mathtt{cs}_1, \mathtt{cs}_2$, resp.) for each $S \in A$.
- 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 $S \rhd \mathtt{cs}_1$ ($S \rhd \mathtt{cs}_2$, $S \rhd \mathtt{cs}_1, \mathtt{cs}_2$, respectively).

Proposition 7 *Let $S, S' \in \mathtt{D} \cup \mathtt{S}$ with $S \approx S'$. Then*:

1. *$S \rhd \mathtt{cs}_i$ implies $S' \rhd \mathtt{cs}_j$ with $\{i, j\} = \{1, 2\}$;*
2. *if S is making progress, then also S' is making progress.*

Proof Item 1 follows from iterative application of Proposition 5 and from the fact that $\mathtt{cs}_i \in v$ implies $\mathtt{cs}_j \in \mathscr{S}(v)$ for $\{i, j\} = \{1, 2\}$. Moreover, since $S \in \mathtt{R}_i$ ($S \in \mathtt{R}_{1,2}$) implies $S' \in \mathtt{R}_j$ with $\{i, j\} = \{1, 2\}$ ($S' \in \mathtt{R}_{1,2}$ respectively), Item 2 follows directly from Item 1.

We now sketch how to prove that all processes in $\mathtt{D} \cup \mathtt{S}$ are making progress; more details can be found in [5, Appendix]. In principle, we construct for each process $D \in \mathtt{D}$ the transition system of those processes and transitions encountered in f-steps from D; then, we read off from this transition system that the actions $\mathtt{cs}_i$ pending for D are indeed performed in every f-execution from D. The basis for the latter is the following simple observation: if for each f-step from some D_y of the form $D_y \xrightarrow{1} R \xrightarrow{v} S \in \tilde{\mathbb{P}}_{\ell 1}$ we have either $\mathtt{cs}_i \in v$ or $S \rhd \mathtt{cs}_i$, then we have $D_y \rhd \mathtt{cs}_i$.

Figure 1 shows the respective transition system for D_8 (states in $\mathtt{D} \cup \mathtt{S}$ are grey); to increase readability, we omit τ-actions in the transition labels. Applying the above observation, we see that $\mathtt{cs}_1 \in v$ for each suitable v, and thus we have $D_8 \rhd \mathtt{cs}_1$. Note that, if only $\mathtt{cs}_1$ were pending for D_8, we would not have to follow the transitions from Q_{23} since $\mathtt{cs}_1$ was already performed when Q_{23} is reached; thus, we would not generate Q_{25} and all subsequent processes. Below, we will indicate a state where we stopped building the transition system by a dashed arrow.

Collecting statements like $D_8 \rhd \mathtt{cs}_1$, we can also deal with more complicated cases. As an example, if D_8, D_9, D_{24}, D_{25}, D_{26} and D_{27} have already been proven to imply the execution of the action $\mathtt{cs}_1$, then we can prove that $D_5 = ((P_{11} \parallel P_2) \parallel_B [\mathsf{PV}(\mathit{true}, \mathit{false}, 1)])[\Phi_B]$ is making progress; cf. the respective transition system for D_5 in Fig. 2.

Starting from D_5, an f-step either runs through Q_9 and is thus clearly not relevant, or it leads to D_8, D_9, D_{24}, D_{25}, D_{26} or to D_{27}. Since D_8, D_9, D_{24}, D_{25}, D_{26} and D_{27} imply the

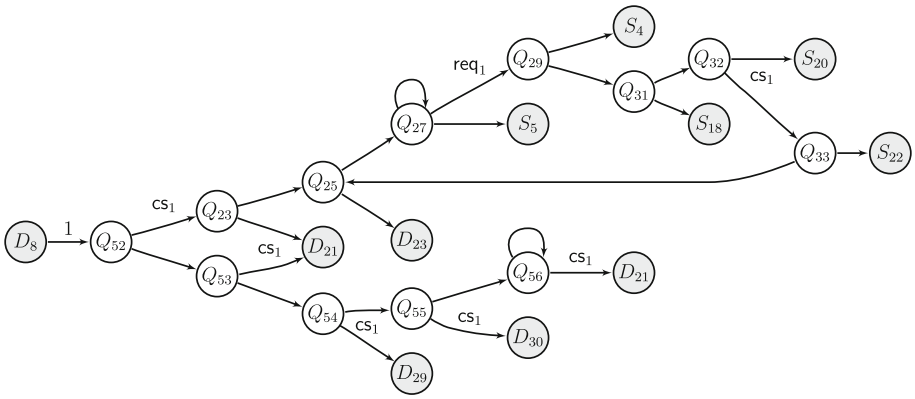

Fig. 1 Fair-steps of D_8

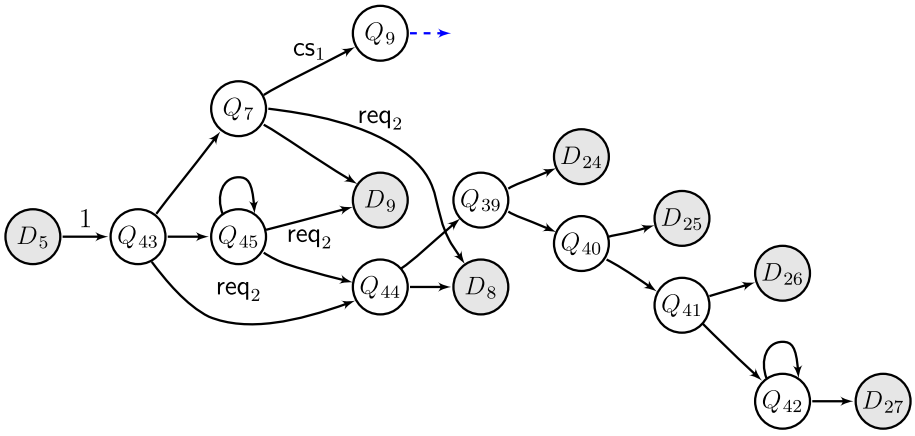

Fig. 2 Fair-steps of D_5

execution of the action cs_1, we can conclude that D_5 implies the execution of cs_1 and, since $D_5 \in \mathtt{R}_1$, that this fair-reachable process is making progress.

Proposition 8 *All processes in* $\mathtt{D} \cup \mathtt{S}$ *are making progress.*

From this proposition, we can conclude:

Proposition 9 *Each f-execution from Dekker*[PV] *is live.*

Proof Assume, by contradiction, that there exists an f-execution from *Dekker*[PV]

$$Dekker[\mathsf{PV}] = P_0 \overset{1}{\longmapsto} Q_0 \overset{v_0}{\longmapsto} P_1 \overset{1}{\longmapsto} Q_1 \overset{v_1}{\longmapsto} P_2 \ldots$$

which is not live. By Definition 16, there exists $j \in \mathbb{N}_0$ such that $P_0 \overset{1}{\longmapsto} Q_0 \overset{v_0}{\longmapsto} P_1 \overset{1}{\longmapsto} Q_1 \overset{v_1}{\longmapsto} P_2 \ldots \overset{v_{j-1}}{\longmapsto} P_j \overset{1}{\longmapsto} Q_j \overset{w_1}{\longmapsto} Q \overset{\mathtt{req}_i}{\longmapsto} Q' \overset{w_2}{\longmapsto} P_{j+1} \overset{1}{\longmapsto} Q_{j+1} \overset{v_{j+1}}{\longmapsto} P_{j+2} \ldots$ where $v_j = w_1 \mathtt{req}_i w_2$ and cs_i does not occur in $w_2 v_{j+1} v_{j+2} \ldots$. Thus, cs_i is pending for P_{j+1} and it does not occur in the f-execution $P_{j+1} \overset{1}{\longmapsto} Q_{j+1} \overset{v_{j+1}}{\longmapsto} P_{j+2} \ldots$. This implies that $P_{j+1} \in \mathtt{D} \cup \mathtt{S}$ is not making progress, contradicting Lemma 8.

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

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

6 Conclusion

In previous work, we have characterised weakly fair traces (w.r.t. actions, components resp.) in two variants of the timed process algebra PAFAS. Here, we demonstrated that something useful can be done with our approach by studying the liveness property of Dekker's MUTEX algorithm: in the action version, the property is violated; this can be explained by the observation that read actions for the same variable can block each other. In the component version, liveness holds; for proving this, we developed some ideas to make the check more efficient.

In future work, an algorithm for such checks should be implemented efficiently. E.g. one could make the concept of permanently lazy component more flexible and consider to make choices and action prefixes lazy; the general strategy would be to distinguish input and output actions, and to make input actions and choices between them lazy. Note that the action-based version of PAFAS could just as well be described without the function clean; as we have pointed out, clean was specifically introduced to have a nicer presentation in relation to fairness. At the same time, clean has a similar effect as the permanently lazy components: due to clean, we do not distinguish between processes that only differ in irrelevant urgencies of actions, so the state space gets smaller.

Currently, we are working on extending the action-oriented approach with specific read actions that do not block other actions. We have actually developed two versions, where one is simpler while the other is more flexible and needs a two-level transition system for the functional behaviour. In the future, we will compare the expressiveness of these two versions, PAFAS and PAFAS^{c}; observe that we are dealing here with weak fairness only, and not with general fairness constraints that are expressible with Büchi automata.

We are confident that our approach can be adapted to other process algebras, e.g. to CCS parallel composition. Actually, CCS has been extended with upper time bounds in [11] and provided with a faster-than relation of bisimulation type; this is closely related to our approach, which in contrast is testing-based. The essential feature for a successful adaptation presumably is that concurrency can be expressed explicitly; actually, our work is based on earlier studies using Petri nets [14]. It is not clear to us how our ideas could be developed in an automata-based setting like timed automata.

Finally, it would be interesting to make the check of liveness properties compositional. For a respective precongruence result, we would consult [14] where a precongruence w.r.t. parallel composition for fair traces is presented.

Acknowledgments We thank the anonymous referees for their helpful comments.

References

1. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: a semantics-based tool for the verification of concurrent systems. In: Proceedings of ACM Transaction on Programming Languages and Systems, vol. 15 (1993)
2. Corradini, F., Di Berardini, M.R., Vogler, W.: PAFAS at work: comparing the worst-case efficiency of three buffer implementations. In: Proceedings of 2nd Asia-Pacific Conference on Quality Software, APAQS 2001, pp. 231–240, IEEE (2001)

3. Corradini, F., Di Berardini, M.R., Vogler, W.: Fairness of components in system computations. Theor. Comput. Sci. **356**(3), 291–324 (2006)
4. Corradini, F., Di Berardini, M.R., Vogler, W.: Fairness of actions in system computations. Acta Inform. **43**, 73–130 (2006)
5. Corradini, F., Di Berardini, M., Vogler, W.: Liveness of a MUTEX algorithm in a fair process algebra. Tech. Rep. 2008-03, University of Augsburg. http://www.Informatik.Uni-Augsburg.DE/forschung/reports/ (2008)
6. Corradini, F., Vogler, W.: Measuring the performance of asynchronous systems with PAFAS. Theor. Comput. Sci. **335**, 187–213 (2005)
7. Corradini, F., Vogler, W., Jenner, L.: Comparing the worst-case efficiency of asynchronous systems with PAFAS. Acta Inform. **38**, 735–792 (2002)
8. Costa, G., Stirling, C.: A fair calculus of communicating systems. Acta Inform. **21**, 417–441 (1984)
9. Costa, G., Stirling, C.: Weak and strong fairness in CCS. Inform. Comput. **73**, 207–244 (1987)
10. Francez, N.: Fairness. Springer, Berlin (1986)
11. Lüttgen, G., Vogler, W.: Bisimulation on speed: worst-case efficiency. Inform. Comput. **191**(2), 105–144 (2004)
12. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco (1996)
13. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall International, Englewood Cliffs (1989)
14. Vogler, W.: Efficiency of asynchronous systems, read arcs, and the MUTEX-problem. Theor. Comput. Sci. **275**, 589–631 (2002)
15. Walker, D.J.: Automated analysis of mutual exclusion algorithms using CCS. Form. Asp. Comput. **1**, 273–292 (1989)