Automated Analysis of MUTEX Algorithms with `FASE` *

Federico Buti, Massimo Callisto De Donato, Flavio Corradini, Maria Rita Di Berardini

School of Science and Technology, University of Camerino

{federico.buti, massimo.callisto, flavio.corradini, mariarita.diberardini}@unicam.it

Walter Vogler

Institut für Informatik, Universität Augsburg

vogler@informatik.uni-Augsburg.de

In this paper we study the liveness of several MUTEX solutions by representing them as processes in PAFAS$_s$, a CCS-like process algebra with a specific operator for modelling non-blocking reading behaviours. Verification is carried out using the tool `FASE`, exploiting a correspondence between violations of the liveness property and a special kind of cycles (called *catastrophic cycles*) in some transition system. We also compare our approach with others in the literature. The aim of this paper is twofold: on the one hand, we want to demonstrate the applicability of `FASE` to some concrete, meaningful examples; on the other hand, we want to study the impact of introducing non-blocking behaviours in modelling concurrent systems.

1 Introduction

MUTEX algorithms can exhibit an intricate behaviour and their correctness can be hard to establish, because our intuitive notion of the program flow can be misled by the fact that a shared variable may change from one statement to the other, even if the process we are tracing does not modify it. There are two kinds of properties to verify: the *safety property* that two competing processes are never in their critical sections at the same time, and the *liveness property* that a requesting process will always enter its critical section. The first kind of property can be proven fairly easily because only the static configuration of the system at any time must be taken into account. The liveness property is much more difficult to prove since it usually requires some fairness assumption.

In [8], we have developed the process description language PAFAS, a CCS-like [11] process algebra originally introduced as a tool for evaluating the worst-case efficiency of asynchronous systems. Processes are compared via a variant of the testing approach of De Nicola and Hennessy [12] where tests are test environments (or user behaviours) together with a time bound. A process is embedded into the environment (via parallel composition) and satisfies a (timed) test, if success is reached before the time bound in *every* run of the composed system, i.e. even in the worst case. This gives rise to a *faster-than* preorder relation over processes that is naturally an *efficiency preorder*. In [7] it has been shown that the test-based preorder in [8] can equivalently be defined on the basis of a performance function that gives the worst-case time needed to satisfy any test environment. Whenever the above testing scenario is adapted to a setting where tests belong to a very specific, but often occurring, class of *request-response* user behaviours (processes serving these users receive requests via an *in*-action and provide responses via an *out*-action) this performance function is *asymptotically linear*. This provides us with a quantitative measure of systems performance that measures how fast the system under consideration responds to requests from the environment. In [7] we have also shown how to determine this performance measure

*This work was supported by the PRIN Project 'Paco:Performability-Aware Computing: Logics, Models, and Languages'.

Giovanna D'Agostino, Salvatore La Torre (Eds.):
Proceedings of the Second International Symposium on
"Games, Automata, Logics and Formal Verification" (GandALF 2011)
EPTCS 54, 2011, pp. 45–59, doi:10.4204/EPTCS.54.4

for finite-state processes. This result only holds for request-response processes (i.e. processes that can only perform *in* and *out* as visible actions) that pass certain sanity checks: they must not produce more responses than requests, and they must allow requests and provide responses in finite time. While the first requirement can easily be read off from the transition system, violation of the latter one is characterised as the existence of a special kind of cycles (called *catastrophic cycles*) in a reduced transition system (we remind the reader to [7] for the complete description of such a reduction). Finally, a corresponding tool FASE that allows the automated evaluation of systems performance function has been developed; see [2] for a first informal account.

The notion of timing in PAFAS is strongly related to (weak) *fairness of actions* which requires that an action must be performed whenever it is enabled continuously in a run. We have shown that each everlasting (or non-Zeno) timed process execution is fair and vice versa, where fairness is defined in an intuitive but complicated way in the spirit of [9, 10]. In fact, we have proven this correspondence for fairness of actions and, with a modified notion of timing, for fairness of components. These characterisations have been used in [5] to prove that Dekker's algorithm is live under the assumption of fairness of components but not under the assumption of fairness of actions. This result can be improved by means of suitable assumptions about the hardware, namely we must assume that reading a value from a storage cell is non-blocking; to model this we have introduced specific reading prefixes for PAFAS in [6].

Here, we add reading in the form of a read-set prefix $\{a_1,\ldots,a_n\} \triangleright P$ (the new process description language is called PAFAS$_s$) which behaves as P but, like a variable or a more complex data structure, can also be read with actions in the set $\{a_1,\ldots,a_n\}$. Since being read does not change the state, each action a_i ($i = 1,\ldots,n$) can be performed repeatedly until the execution of some ordinary action of P .

A first key property of non-blocking actions is that they have a direct impact on timed behaviour of concurrent systems (see the examples at the end of Section 2). They are also an important feature for proving the liveness of MUTEX solutions under the assumption of weak fairness of actions. Indeed, one result in [6] shows that Dekker's algorithm is live when assuming fairness of actions, provided we regard as non-blocking the reading of a variable as well as its writing in the case that the written value equals the current one. It had long been an open problem how to achieve such a result in a process algebra (see e.g. [13]). In [6] we have also discovered an interesting connection between liveness of MUTEX algorithms and catastrophic cycles; we have shown that violations of the liveness property can be traced back to catastrophic cycles of a suitably modified process (cf. Section 3). Even though FASE was originally developed for automatically checking whether a process of (original) PAFAS has a catastrophic cycle, it has been recently adapted to a setting with reading actions. This has opened the way to check automatically the liveness property for MUTEX algorithms.

In this paper we use FASE to study the liveness of four MUTEX solutions–Peterson's, Lamport's, Dijkstra's and Knuth's algorithms (see [13] and references therein)–under the assumption of fairness of actions. Our aim is twofold: we want to show the applicability of FASE to concrete, meaningful examples, but also to stress the impact of introducing non-blocking actions in PAFAS (and in general in modelling concurrent systems). We prove that Peterson is live provided we regard the reading of a variable as a non-blocking action. We also show that the liveness of Dijkstra and Knuth cannot be ensured even if (as in [6]) we consider as non-blocking the reading of a variable and its writing in the case the written value equals the current one. With the same assumption on program variables, we finally prove that Lamport (which is not symmetric) is live for just one of the two competing processes, i.e. it is not live.

To even more emphasize the role of non-blocking reading in proving liveness property, we have implemented some ideas taken from [13] that describe how fairness can be assumed in a CCS setting in order to enable a proof of liveness. At the time of writing, these ideas could not be expressed for the use of the Concurrency Workbench [4], but this is now possible within newer tools like the Concurrency

Workbench of the New Century [3]. A comparison of the results provided by the two approaches shows that the liveness of Dekker's and Peterson's algorithms strongly depends on the liveness of the hardware. This is exactly the sort of consideration for which non-blocking actions provide a formal treatment.

We proceed as follows: In Section 2 we recall PAFAS$_s$, its timed operational semantics and the correspondence between fair traces and everlasting timed computations. In Section 3 we introduce the four algorithms and provide our results. Finally, in Section 4 we compare our approach with that in [13].

2 A process algebra for describing reading behaviours

PAFAS [8] is a CCS-like process description language [11] (with a *TCSP*-like parallel composition), where actions are atomic and instantaneous, but have associated an upper time bound (either 0 or 1, for simplicity) as a maximal delay for their execution. As shown in [8], these upper time bounds can be used to evaluate efficiency, but they do not influence functionality (which actions are performed); so compared with CCS also PAFAS treats the full functionality of asynchronous systems. In [6], PAFAS has been extended with a new operator $\triangleright$ to represent non-blocking behaviour of processes. Intuitively, $\{\alpha_1,\dots,\alpha_n\}\triangleright P$ models a process like a variable or a more complex data structure that behaves as P but can additionally be read with $\alpha_1,\dots,\alpha_n$: since being read does not change the state, each action α_i can be performed repeatedly without blocking a synchronization partner as described below. We use the following notation: $\mathbb{A}$ is an infinite set of *visible actions*; the additional action τ represents a internal activity, unobservable for other components, and $\mathbb{A}_\tau = \mathbb{A}\cup\{\tau\}$. Elements of $\mathbb{A}$ are denoted by $a,b,c,\dots$ and those of $\mathbb{A}_\tau$ by $\alpha,\beta,\dots$. 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 $\underline{\mathbb{A}}_\tau = \{\underline{a}\,|\,a\in\mathbb{A}\}\cup\{\underline{\tau}\}$ and is ranged over by $\underline{\alpha},\underline{\beta},\dots$. Elements of $\mathbb{A}_\tau\cup\underline{\mathbb{A}}_\tau$ are ranged over by μ and ν. We also assume that, for any $\alpha\in\mathbb{A}_\tau$, when time elapses $\underline{\underline{\alpha}} = \underline{\alpha}$. $\mathscr{X}$ (ranged over by $x,y,z,\dots$) is the set of process variables, used for recursive definitions. $\Phi : \mathbb{A}_\tau \to \mathbb{A}_\tau$ is a *general relabelling function* if the set $\{\alpha\in\mathbb{A}_\tau\,|\,\emptyset\neq\Phi^{-1}(\alpha)\neq\{\alpha\}\}$ 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 $\alpha\notin A$.

Below, initial processes are just processes of a standard process algebra extended with $\triangleright$, while general processes are those reachable from the initial ones according to the operational semantics. The set $\tilde{\mathbb{S}}_1$ of *initial (timed) process terms P* and $\tilde{\mathbb{S}}$ of (general) *(timed) process terms Q* are generated by:

$$P ::= \mathsf{nil} \,\big|\, x \,\big|\, \alpha.P \,\big|\, \{\alpha_1,\dots,\alpha_n\}\triangleright P \,\big|\, P+P \,\big|\, P\,\|_A\,P \,\big|\, P[\Phi] \,\big|\, \mathsf{rec}\,x.P$$
$$Q ::= P \,\big|\, \underline{\alpha}.P \,\big|\, \{\mu_1,\dots,\mu_n\}\triangleright Q \,\big|\, Q+Q \,\big|\, Q\,\|_A\,Q \,\big|\, Q[\Phi]$$

where nil is a constant, $x\in\mathscr{X}$, $\alpha\in\mathbb{A}_\tau$, Φ is a general relabelling function and $A\subseteq\mathbb{A}$ possibly infinite; $\{\alpha_1,\dots,\alpha_n\}$ and $\{\mu_1,\dots,\mu_n\}$ are (finite and nonempty) subsets of $\mathbb{A}_\tau$ and $\mathbb{A}_\tau\cup\underline{\mathbb{A}}_\tau$, resp. We assume that the latter kind of read-sets can only contain a copy (either lazy or urgent) of each action α, i.e. $\{\mu_1,\dots,\mu_n\}$ cannot contain both α *and* $\underline{\alpha}$ for any $\alpha\in\mathbb{A}_\tau$. By the operational semantics, terms not satisfying this property are not reachable from initial ones anyway. A process term is *closed* if every variable x is bound by the corresponding $\mathsf{rec}\,x$-operator; the set of closed timed process terms in $\tilde{\mathbb{P}}$ and $\tilde{\mathbb{P}}_1$, simply called *processes* and *initial processes* resp., is denoted by $\mathbb{P}$ and $\mathbb{P}_1$ resp.

nil is the Nil-process: it cannot perform any action but can let time pass without limits. $\alpha.P$ and $\underline{\alpha}.P$ is action-prefixing known from CCS. Process $\alpha.P$ performs α within time 1; i.e. it can perform α immediately and evolve to P (as usual in CCS), or let one time unit pass and become $\underline{\alpha}.P$. In this latter case, α cannot be further delayed (i.e. it must occur or be deactivated) unless $\underline{\alpha}.P$ has to wait for a synchronisation on $\alpha\neq\tau$. Our processes are *patient*: as a stand-alone process $\underline{a}.P$ has no reason to wait, but as a component of a larger system, e.g. $\underline{a}.P \,\|_{\{a\}}\, a.\mathsf{nil}$, it can wait for a synchronisation on a; this can take up to time 1 since component $a.\mathsf{nil}$ can idle so long. $\{\mu_1,\ldots,\mu_n\} \triangleright Q$ can perform actions from $\{\mu_1,\ldots,\mu_n\}$ without changing state (including urgencies and, hence, the syntax of the term itself), and the actions of Q in the same way as Q, i.e. the read-set is removed after such an action. $Q_1 + Q_2$ is a nondeterministic choice between two conflicting processes Q_1 and Q_2. Q_1 and Q_2 run in parallel in $Q_1 \|_A Q_2$ and have to synchronize on all actions from A. $P[\Phi]$ behaves as P but with actions changed according to Φ. $\mathsf{rec}\,x.P$ models a recursive definition; we often use equations to define recursive processes.

Functional and temporal behaviour of PAFAS$_s$ processes. We first introduce the transitional semantics describing the functional behaviour of PAFAS$_s$ processes, i.e. which actions they can perform.

Definition 2.1 (*functional operational semantics*) Let $Q \in \tilde{\mathbb{S}}$ and $\alpha \in \mathbb{A}_\tau$. The SOS-rules defining the transition relation $\xrightarrow{\alpha} \subseteq (\tilde{\mathbb{S}} \times \tilde{\mathbb{S}})$ (the *action transitions*) are given in Table 1[1]. As usual, we write $Q \xrightarrow{\alpha} Q'$ if $(Q,Q') \in \xrightarrow{\alpha}$ and $Q \xrightarrow{\alpha}$ if there exists a $Q' \in \tilde{\mathbb{S}}$ such that $(Q,Q') \in \xrightarrow{\alpha}$. Similar conventions will apply later on. We also define the set of the *activated* or enabled actions to be the set of all α such that $Q \xrightarrow{\alpha}$.

$$\textsc{Pref}_s\ \frac{\mu \in \{\alpha, \underline{\alpha}\}}{\mu.P \xrightarrow{\alpha} P} \qquad \textsc{Sum}_s\ \frac{Q_1 \xrightarrow{\alpha} Q'}{Q_1 + Q_2 \xrightarrow{\alpha} Q'} \qquad \textsc{Read}_{s1}\ \frac{\mu_i \in \{\alpha, \underline{\alpha}\}}{\{\mu_1,\ldots,\mu_n\} \triangleright Q \xrightarrow{\alpha} \{\mu_1,\ldots,\mu_n\} \triangleright Q}$$

$$\textsc{Read}_{s2}\ \frac{Q \xrightarrow{\alpha} Q'}{\{\mu_1,\ldots,\mu_n\} \triangleright Q \xrightarrow{\alpha} Q'} \qquad \textsc{Par}_{s1}\ \frac{\alpha \notin A,\ Q_1 \xrightarrow{\alpha} Q_1'}{Q_1 \|_A Q_2 \xrightarrow{\alpha} Q_1' \|_A Q_2} \qquad \textsc{Par}_{s2}\ \frac{\alpha \in A,\ Q_1 \xrightarrow{\alpha} Q_1',\ Q_2 \xrightarrow{\alpha} Q_2'}{Q_1 \|_A Q_2 \xrightarrow{\alpha} Q_1' \|_A Q_2'}$$

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

Table 1: Functional behaviour of PAFAS$_s$ processes

Rules in Table 1 are quite standard. Timing can be disregarded in $\textsc{Pref}_s$: when an action is performed, one cannot see whether it was urgent or not, and thus $\underline{\alpha}.P \xrightarrow{\alpha} P$; furthermore, component $\alpha.P$ has to act *within* time 1, i.e. it can also act immediately, giving $\alpha.P \xrightarrow{\alpha} P$. Rules $\textsc{Read}_{s1}$ and $\textsc{Read}_{s2}$ say that $\{\mu_1,\ldots,\mu_n\} \triangleright Q$ can either repeatedly perform one of its non-blocking actions or evolve as Q. Other rules are as expected; symmetric rules have been omitted. Actually, the above SOS-rules describe reading in a sensible way only under some syntactic restrictions, cf. [6]. All the example processes we consider here meet these restrictions.

We now define the refusal traces of a term $Q \in \tilde{\mathbb{S}}$. Intuitively a refusal trace records, along a computation, which actions process Q can perform ($Q \xrightarrow{\alpha} Q'$, $\alpha \in \mathbb{A}_\tau$) and which actions Q can refuse to perform when time elapses ($Q \xrightarrow{X}_r Q'$, $X \subseteq \mathbb{A}$). $Q \xrightarrow{X}_r Q'$ is called a (partial) *time-step*. The actions listed in X are not urgent; hence Q is justified in not performing them, but performing a time step instead. This time step is partial because it can occur only in contexts that can refuse the actions not in X. If $X = \mathbb{A}$ then Q is fully justified in performing this time-step; i.e., Q can perform it independently of the environment. In such a case, we say that Q performs a *1-step* written $Q \xrightarrow{1} Q'$; moreover we often write $\underline{Q}$ (the urgent version of Q) instead of Q'. To provide the reader with a better intuition we observe that any Q can perform a 1-step whenever it can refuse to perform, because not urgent, all its activated actions. In the next definition, $\mathscr{U}(\{\mu_1,\ldots,\mu_n\}) = \{\alpha \mid \mu_i = \underline{\alpha} \text{ for some } i \in [1,n]\}$ is the set of urgent actions in $\{\mu_1,\ldots,\mu_n\}$.

[1] We do here without clean and unmark, used e.g. in [5] to get a closer relationship between states of untimed fair runs and timed non-Zeno runs. They do not change the behaviour (up to an injective bisimulation) and would complicate the setting.

Definition 2.2 (*refusal transitional semantics*) The SOS-rules in Table 2 define $\xrightarrow{X}_r \subseteq (\tilde{\mathbb{S}} \times \tilde{\mathbb{S}})$ where $X \subseteq \mathbb{A}$.

$$\textsc{Nil}_t\ \frac{}{\mathsf{nil} \xrightarrow{X} \mathsf{nil}} \qquad \textsc{Pref}_{t1}\ \frac{}{\alpha.P \xrightarrow{X}_r \underline{\alpha}.P} \qquad \textsc{Pref}_{t2}\ \frac{\alpha \notin X \cup \{\tau\}}{\underline{\alpha}.P \xrightarrow{X}_r \underline{\alpha}.P} \qquad \textsc{Sum}_t\ \frac{Q_i \xrightarrow{X}_r Q_i' \text{ for } i=1,2}{Q_1 + Q_2 \xrightarrow{X}_r Q_1' + Q_2'}$$

$$\textsc{Read}_t\ \frac{Q \xrightarrow{X}_r Q',\ \mathscr{U}(\{\mu_1,\dots,\mu_n\}) \cap (X \cup \{\tau\}) = \emptyset}{\{\mu_1,\dots,\mu_n\} \triangleright Q \xrightarrow{X}_r \underline{\{\mu_1,\dots,\mu_n\}} \triangleright Q'} \qquad \textsc{Rel}_t\ \frac{Q \xrightarrow{\Phi^{-1}(X \cup \{\tau\}) \backslash \{\tau\}}_r Q'}{Q[\Phi] \xrightarrow{X}_r Q'[\Phi]}$$

$$\textsc{Par}_t\ \frac{Q_i \xrightarrow{X}_r Q_i' \text{ for } i=1,2, X \subseteq (A \cap (X_1 \cup X_2)) \cup ((X_1 \cap X_2) \backslash A)}{Q_1 \|_A Q_2 \xrightarrow{X}_r Q_1' \|_A Q_2'} \qquad \textsc{Rec}_t\ \frac{Q\{\mathsf{rec}\,x.Q/x\} \xrightarrow{X}_r Q'}{\mathsf{rec}\,x.Q \xrightarrow{X}_r Q'}$$

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

Rule $\textsc{Pref}_{t1}$ says that a process $\alpha.P$ can let time pass and can refuse to perform any action, while rule $\textsc{Pref}_{t2}$ says that a process $\underline{\alpha}.P$, can let time pass but action α cannot be refused. Process $\underline{\tau}.P$ cannot let time pass and cannot refuse any action; in any context, $\underline{\tau}.P$ has to perform τ before time can pass further. Rule $\textsc{Par}_t$ defines which actions a parallel composition can refuse during a time-step. The intuition is that $Q_1 \|_A Q_2$ can refuse an action α if either $\alpha \notin A$ (Q_1 and Q_2 can do α independently) and both Q_1 and Q_2 can refuse α, or $\alpha \in A$ (Q_1 and Q_2 are forced to synchronise on α) and at least one of them can refuse the action, i.e. can delay it. Thus, an action in a parallel composition is urgent (cannot be further delayed) only when all synchronising 'local' actions are urgent. Rule $\textsc{Read}_t$ says that $\{\mu_1,\dots,\mu_n\} \triangleright Q$ can refuse the same actions as Q provided these are not urgent in $\{\mu_1,\dots,\mu_n\}$; moreover, as for the action-prefixing, process $\{\mu_1,\dots,\mu_n\} \triangleright Q$ cannot let time pass and cannot refuse any action, whenever one of the urgent actions in $\{\mu_1,\dots,\mu_n\}$ is a τ. Other rules are as expected. Again symmetric rules have been omitted.

In [8], it is shown that inclusion of refusal traces characterises a testing-based faster-than relation that compares processes w.r.t. their worst-case efficiency. In this sense, e.g. $P = \{a\} \triangleright b.\mathsf{nil}$ is faster than the functionally equivalent $P' = \mathsf{rec}\,x.\,a.x + b.\mathsf{nil}$, since only the latter has the refusal traces $1a(1a)^*$. After $1a$, P' returns to itself (recursion unfolding creates fresh a and b); intuitively, b is disabled during the occurrence of a, so a and also b can be delayed again. In contrast, after a 1-step and any number of a's, P turns into $\{\underline{a}\} \triangleright \underline{b}.\mathsf{nil}$ and no further 1-step is possible; read actions do not block or delay other activities, they make processes faster. If a models the reading of a value stored by P or P' and two parallel processes want to read it, this should take at most time 1 in a setting with non-blocking reads. And indeed, whereas $P' \|_{\{a\}} (a.\mathsf{nil} \|_\emptyset a.\mathsf{nil})$ has the refusal trace $1a1a$, this behaviour is not possible for $P \|_{\{a\}} (a.\mathsf{nil} \|_\emptyset a.\mathsf{nil})$ since, when performing $1a$, this evolves into e.g. $\{\underline{a}\} \triangleright \underline{b}.\mathsf{nil} \|_{\{a\}} (\mathsf{nil} \|_\emptyset \underline{a}.\mathsf{nil})$, and then 1 is not possible.

Another application of refusal traces is the modelling of *weak fairness of actions*. Weak fairness requires that an action must be performed whenever continuously enabled in a run. Thus, a run from P above with infinitely many a's is not fair; the read action does not block b or change the state, so the same b is always enabled but never performed. In contrast, if P' performs a, a fresh b is created; in conformance to [9], a run from P' with infinitely many a's is fair. In [6], generalising [5], fair traces for PAFAS$_s$ are first defined in an intuitive, but very complex fashion in the spirit of [9, 10] and then characterised: *they are the sequences of visible actions occurring in transition sequences with infinitely many 1-steps*. Due to lack of space, we cannot properly formulate this as a theorem, but take it as a **definition** of *fair traces* instead. With this, infinitely many a's are a fair trace of P' since it can repeat $1a$ indefinitely, but the fair traces of finite-state P are those that end with b. We use this definition of fair traces to study liveness property of MUTEX solutions we consider in the next section.

For request-response processes the transition system (built according to Def. 2.1 and 2.2) must be reduced as described in [7]; a cycle in the resulting system is catastrophic if it contains (at least) one time step but no *in*- or *out*-transition.

3 Liveness property of MUTEX algorithms

In this section we use the approach of [6] to study the liveness of four different MUTEX solutions: Peterson's, Lamport's, Dijkstra's and Knuth's algorithm. We first translate the algorithms into PAFAS$_s$ and then use `FASE` to automatically decide whether each of them is live or not. Negative results are discussed by means of counterexamples, i.e. fair violating traces which are built from catastrophic cycles detected with `FASE`. The results of this section are collected in Table 3.

Peterson's algorithm There are two processes $\mathtt{P}_1$ and $\mathtt{P}_2$, two Boolean-valued variables b_1 and b_2, whose initial value is false, and a variable k, which takes values in $\{1,2\}$ and whose initial value is arbitrary. The b_i variables are "request" variables and k is a "turn" variable: b_i is true if $\mathtt{P}_i$ is requesting entry to its critical section and k is i if it is $\mathtt{P}_i$'s turn to enter its critical section. Only $\mathtt{P}_i$ writes b_i, but both processes read it. Process $\mathtt{P}_i$ (with $i = 1,2$) is described as follows; j is the index of the other process:

PETERSON()
while true
  do ⟨non-critical section⟩;
    $b_i \leftarrow true$; $k \leftarrow j$;
    while b_j and $k = j$ **do skip**;
    ⟨critical section⟩;
    $b_i \leftarrow false$;

In our translation of the algorithm into PAFAS$_s$, we use essentially the same coding as Walker in [13]. Each variable is represented as a family of processes. For example, the process $\mathsf{B}_1(f)$ denotes the variable b_1 with value false. The *sort* of $\mathsf{B}_1(f)$ (i.e. the set of actions it can ever perform) is $\{b_1rf, b_1rt, b_1wf, b_1wt\}$ Unlike [13], we model the actions that correspond to the reading of a variable (e.g. b_1rt and b_2rt) as non-blocking. Below, we let $\mathbb{B} = \{f,t\}$ and $\mathbb{K} = \{1,2\}$.

Definition 3.1 (*Peterson's algorithm*) Let $i \in \{1,2\}$. Program variables are represented as follows:

$$\mathsf{B}_i(f) = \{b_irf\} \triangleright (b_iwf.\mathsf{B}_i(f) + b_iwt.\mathsf{B}_i(t)) \qquad \mathsf{K}(1) = \{kr1\} \triangleright (kw1.K(1) + kw2.\mathsf{K}(2))$$
$$\mathsf{B}_i(t) = \{b_irt\} \triangleright (b_iwt.\mathsf{B}_i(t) + b_iwf.\mathsf{B}_i(f)) \qquad \mathsf{K}(2) = \{kr2\} \triangleright (kw1.K(1) + kw2.\mathsf{K}(2))$$

Given $b_1, b_2 \in \mathbb{B}$, $k \in \mathbb{K}$, we define $\mathsf{PV}(b_1,b_2,k) = (\mathsf{B}_1(b_1) \|_\emptyset \mathsf{B}_2(b_2)) \|_\emptyset \mathsf{K}(k)$. Processes $\mathtt{P}_1$ and $\mathtt{P}_2$ are represented by the following PAFAS$_s$ processes: the actions $\mathtt{req}_i$ and cs_i indicate the request to enter and the execution of the critical section by the process $\mathtt{P}_i$.

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

Since no process should be forced to request by the fairness assumption, P_i has the alternative of an internal move, i.e. staying in its non-critical section. Peterson's algorithm is defined to be the PAFAS$_s$ process $\mathsf{Peterson} = ((P_1 \|_\emptyset P_2) \|_B \mathsf{PV}(f,f,1))/B$; here (and in the following) B is the set of all actions except $\mathtt{req}_i$ and cs_i ($i = 1,2$). A MUTEX algorithm like Peterson's satisfies *liveness* if, in every fair trace, each $\mathtt{req}_i$ is eventually followed by the respective cs_i.

We now show how to modify the process $\mathsf{Peterson}$ such that it is live under the assumption of fairness of actions iff the modified process, that we call $\mathsf{Peterson}_{io}$, does not have catastrophic cycles. Observe that FASE only accepts request-response behaviours (having only *in* and *out* as visible actions) as input and, hence, it cannot be applied directly. Moreover, $\mathsf{Peterson}$ can perform a 1-step followed by the two internal actions of P_1 and P_2 (see Def. 3.1) giving a catastrophic cycle which is not relevant for the liveness property. So, we modify $\mathsf{Peterson}$ as follows: we first change $\mathtt{req}_1$ and cs_1 into τ's and $\mathtt{req}_2$ and cs_2 into *in* and *out*, resp.; we finally delete the τ summand of P_2. As in [6] (see Theorem 8.2[2]), we can prove that $\mathsf{Peterson}_{io}$ does not have catastrophic cycles iff each request from process $\mathtt{P}_2$ will eventually be satisfied along fair traces, i.e. iff $\mathsf{Peterson}$ is live for process $\mathtt{P}_2$ under the assumption of fairness of actions. The liveness of $\mathsf{Peterson}$ follows by the symmetry of the algorithm. In case of non-symmetric algorithms, as e.g. Lamport, the liveness for processes $\mathtt{P}_1$ and $\mathtt{P}_2$ must be proven separately. Since FASE has shown that $\mathsf{Peterson}_{io}$ does not have catastrophic cycles, our first result is:

Proposition 3.2 $\mathsf{Peterson}$ *is live under the assumption of fairness of actions.*

We now consider $\mathsf{Peterson}'$, a slightly different specification of $\mathsf{Peterson}$ where all actions – including the reading of program variables – are ordinary actions. E.g., in this version, we define $\mathsf{B}_i(f) = b_i rf.\mathsf{B}_i(f) + b_i wf.\mathsf{B}_i(f) + b_i wt.\mathsf{B}_i(t)$. Then, $\mathsf{Peterson}'$ can be defined as in Def. 3.1.

Proposition 3.3 $\mathsf{Peterson}'$ *is not live under the assumption of fairness of actions.*

Proof: FASE shows that $\mathsf{Peterson}'_{io}$ has catastrophic cycles as, e.g., those in the next examples. □

The following example shows a timed computation along which both processes $\mathtt{P}_1$ and $\mathtt{P}_2$ get stuck after a request. To ease understanding, we leave the actions on program variables visible, i.e. we consider a timed computation of $P = (P_1 \|_\emptyset P_2) \|_B \mathsf{PV}(f,f,1)$. Indeed, by the operational semantics, we know that P behaves as $\mathsf{Peterson}'$ as long as we rename actions in B with τ. We will proceed in this fashion later on in this section. Furthermore, we write $\mathsf{PV}(\underline{t},\underline{t},1)$ and $\mathsf{PV}(t,\underline{t},1)$ to abbreviate $(\underline{\mathsf{B}_1(t)} \|_\emptyset \underline{\mathsf{B}_2(t)}) \|_\emptyset \mathsf{K}(1)$ and $(\mathsf{B}_1(t) \|_\emptyset \underline{\mathsf{B}_2(t)}) \|_\emptyset \mathsf{K}(1)$, resp. In general, we underline a value to denote the urgent version of the PAFAS$_s$ process that represents the corresponding variable.

Example 3.4 Consider the following timed computation from P.

$$\begin{array}{rcll}
P & = & (P_1 \|_\emptyset P_2) \|_B \mathsf{PV}(f,f,1) & \xrightarrow{\mathtt{req}_1\, b_1wt\, kw2\, \mathtt{req}_2\, b_2wt\, kw1} \\
 & & (P_{11} \|_\emptyset P_{21}) \|_B \mathsf{PV}(t,t,1) & \xrightarrow{b_2rt\, b_1rt} \\
R & = & (P_{12} \|_\emptyset P_{22}) \|_B \mathsf{PV}(t,t,1) & \xrightarrow{1} \\
\underline{R} & = & (\underline{P_{12}} \|_\emptyset \underline{P_{22}}) \|_B \underline{\mathsf{PV}(t,t,1)} & \xrightarrow{kr1} \\
 & & (\underline{P_{12}} \|_\emptyset P_{21}) \|_B \mathsf{PV}(\underline{t},\underline{t},1) & \xrightarrow{b_1rt} \\
Q & = & (\underline{P_{12}} \|_\emptyset P_{22}) \|_B \mathsf{PV}(t,\underline{t},1) & \xrightarrow{1} \underline{R}
\end{array}$$

Process R can only perform *kr1* as a synchronisation between $\mathsf{K}(1)$ and either P_{12} or P_{22}; after the first 1-step, this action becomes urgent. Once in $\underline{R}$, we perform *kr1* and $\underline{\mathsf{K}}(1)$ evolves into $\mathsf{K}(1)$ which can delay *kr1*. As a consequence, Q can refuse to perform *kr1* and, since this is its only activated action, $Q \xrightarrow{1} \underline{R}$. The execution sequence $\mathsf{Peterson}' = P/B \xrightarrow{\mathtt{req}_1\, \tau^2\, \mathtt{req}_2\, \tau^4} R/B \xrightarrow{\tau^2} R/B \ldots$ is fair but not live since

[2] The proof of Theorem 8.2 we provide in [6] is partly independent from the specific algorithm we were analysing, i.e. Dekker's algorithm, and it can be easily adapted to all the algorithms we consider in this paper. From now on, we freely use the correspondence between liveness and catastrophic cycles without explicitly proving it. In the following, if P is a PAFAS$_s$ process that models a given MUTEX solution, we write P_{io} to denote the process we obtain by changing P as $\mathsf{Peterson}$.

no process will ever enter the critical section; $\underline{R}/B \xrightarrow{\tau^2} Q/B \xrightarrow{1} \underline{R}/B$ corresponds to a catastrophic cycle in the reduced transition system of $\mathsf{Peterson}'_{io}$.

This example describes a scenario where process $\mathtt{P}_1$ will never move because process $\mathtt{P}_2$ repeatedly reads variables k and b_1. There is another fair run where $\mathtt{P}_1$, reading variable b_2, can repeatedly delay and, thus, indefinitely block $\mathtt{P}_2$ that wants to write it. On the contrary, the representation of program variables we use in Def. 3.1 ensures the liveness of the hardware under the assumption of fairness of actions; namely, it ensures that no process can be indefinitely blocked by infinite reading.

Lamport's algorithm There are $n \geq 2$ processes and n Boolean-valued variables b_i ($i = 1 \dots n$), each with initial value false; only $\mathtt{P}_i$ writes b_i, but all the processes can read it. The i-th process is described below:

```
LAMPORT()
var j : integer;
while true
    do ⟨non-critical section⟩;
       b_i ← true;
       for j ← 1 to i − 1
           do if b_j
                 then b_i ← false;
                      while b_j do skip;
                      goto 4;
       for j ← i + 1 to n
           do while b_j do skip;
       ⟨critical section⟩;
       b_i ← false;
```

Now we provide the PAFAS$_s$ specification in case of $n = 2$ processes.

Definition 3.5 (*Lamport's algorithm*) Again we first define the family of PAFAS$_s$ processes representing the program variables. Let $\mathsf{B}_i(f) = \{b_i rf, b_i wf\} \triangleright b_i wt.\mathsf{B}_i(t)$ and $\mathsf{B}_i(t) = \{b_i rt, b_i wt\} \triangleright b_i wf.\mathsf{B}_i(f)$ where $i \in \{1,2\}$. We also define $\mathsf{PV}(b_1,b_2) = \mathsf{B}_1(b_1) \parallel_\emptyset \mathsf{B}_2(b_2)$ where $b_1, b_2 \in \mathbb{B}$.
Processes $\mathtt{P}_1$ and $\mathtt{P}_2$ are represented by:

$$P_1 = \mathtt{req}_1.b_1wt.P_{11} + \tau.P_1 \qquad P_2 = \mathtt{req}_2.b_2wt.P_{21} + \tau.P_2$$
$$P_{11} = b_2rf.P_{12} + b_2rt.P_{11} \qquad P_{21} = b_1rf.P_{23} + b_1rt.b_2wf.P_{22}$$
$$P_{12} = \mathsf{cs}_1.b_1wf.P_1 \qquad P_{22} = b_1rf.b_2wt.P_{21} + b_1rt.P_{22}$$
$$P_{23} = \mathsf{cs}_2.b_2wf.P_2$$

Finally $\mathsf{Lamport} = ((P_1 \parallel_\emptyset P_2) \parallel_B \mathsf{PV}(f,f))/B$.

Note that now we regard as non-blocking not only the reading of a variable but also its writing in case that the written value equals the current one. This kind of re-write does not change the state of the variable and can be thought of as a non-destructive or non-consuming operation (allowing potential concurrent behaviour). This way of accessing a variable is not new. It has been implemented e.g. in area of database. Unlike in *Peterson*'s specification, we make this assumption on the hardware to show that *Lamport*'s algorithm is not live with respect to $\mathtt{P}_2$:

Proposition 3.6 *If we assume fairness of actions,* Lamport *is live for process* $\mathtt{P}_1$ *but* not *for process* $\mathtt{P}_2$.

Proof: Lamport is not live for $\mathtt{P}_2$ because $\mathsf{Lamport}_{io}$ has catastrophic cycles. To prove the other statement, we need symmetric changes; namely, we rename actions $\mathtt{req}_1$ and cs_1 into *in* and *out* resp. and actions $\texttt{req}_2$ and cs_2 into τ; we also delete the τ-summand of process $\mathtt{P}_1$. Since this modified process does not have catastrophic cycles, we conclude that $\mathsf{Lamport}$ is live for process $\mathtt{P}_1$. □

Prop. 3.6 still holds if we use the same representation of program variables as in Def. 3.1, while we lose liveness for $\mathtt{P}_1$ whenever processes representing program variables are those used for $\mathsf{Peterson}'$. Then, while reading variable b_1, process $\mathtt{P}_2$ can forever block the other process that wants to write it. The next example explains why $\mathsf{Lamport}$ is not live for process $\mathtt{P}_2$.

Example 3.7 The following timed computation corresponds to an execution sequence from $\mathsf{Lamport} = L/B$ which is fair but not live since process $\mathtt{P}_2$ never enters its critical section.

$$
\begin{array}{lcll}
L & = & (P_1 \,\|_\emptyset\, P_2) \,\|_B\, \mathsf{PV}(f,f) & \xrightarrow{\texttt{req}_1\ \texttt{req}_2} \\
 & & (b_1wt.P_{11} \,\|_\emptyset\, b_2wt.P_{21}) \,\|_B\, \mathsf{PV}(f,f) & \xrightarrow{b_1wt\ b_2rf} \\
 & & (P_{12} \,\|_\emptyset\, b_2wt.P_{21}) \,\|_B\, \mathsf{PV}(t,f) & \xrightarrow{b_2wt\ b_1rt\ b_2wf} \\
R & = & (P_{12} \,\|_\emptyset\, P_{22}) \,\|_B\, \mathsf{PV}(t,f) & \xrightarrow{1} \\
\underline{R} & = & (\underline{P_{12}} \,\|_\emptyset\, \underline{P_{22}}) \,\|_B\, \underline{\mathsf{PV}(t,f)} & \xrightarrow{\mathsf{cs}_1\ b_1wf} \\
 & & (P_1 \,\|_\emptyset\, \underline{P_{22}}) \,\|_B\, \mathsf{PV}(f,\underline{f}) & \xrightarrow{\texttt{req}_1\ b_1wt\ b_2rf} \\
Q & = & (P_{12} \,\|_\emptyset\, \underline{P_{22}}) \,\|_B\, \mathsf{PV}(t,\underline{f}) & \xrightarrow{1} \underline{R}
\end{array}
$$

R can do either a cs_1- or a b_1rt-action (due to a synchronisation between P_{22} and $\mathsf{B}_1(t)$); both actions become urgent after the first 1-step. Later, we perform cs_1 followed by b_1wf (and, hence, $\underline{\mathsf{B}_1(t)}$ evolves into $\mathsf{B}_1(f)$) which, in turn, is followed by $\texttt{req}_1$ and b_1wt. At this stage, $\mathsf{B}_1(f)$ becomes $\mathsf{B}_1\overline{(t)}$ and Q can refuse to perform its activated actions, again cs_1 and b_1rt, and evolve into $\underline{R}$. Finally, $\underline{R}/B \xrightarrow{\mathsf{cs}_1\,\tau\,\texttt{req}_1\,\tau^2} Q/B \xrightarrow{1} \underline{R}/B$ corresponds to a catastrophic cycle in the reduced transition system of $\mathsf{Lamport}_{io}$. A key observation here is that process $\mathtt{P}_1$, along this cycle, continuously changes the value of b_1 from *true* to *false* and vice versa. Consequently, the PAFAS$_s$ process representing this variable always offers a new instance of b_1rf and b_1rt to its synchronisation partners, and in particular to $\underline{P_{22}}$. So, any possible move of process $\mathtt{P}_2$ can be arbitrarily delayed (and, hence, this process can indefinitely be blocked) even in fair traces. No reasonable assumption about program variables can prevent this unwanted behaviour under weak fairness.

Dijkstra's algorithm This algorithm considers $n \geq 2$ processes that share two Boolean-valued arrays b and c (whose components are initialised to true) and a turn variable k initially chosen in $\{1,2,\ldots,n\}$. The i-th process is described below:

```
DIJKSTRA()
var j : integer;
while true
    do ⟨non-critical section⟩;
       b[i] ← false;
       if k ≠ i
         then c[i] ← true;
              if b[k] then k ← i;
              goto 5;
         else c[i] ← false;
              for j ← 1 to n
                  do if j ≠ i and ¬c[j] then goto 5;
```

⟨critical section⟩;
$c[i] \leftarrow true$;
$b[i] \leftarrow true$;

Again we provide the PAFAS$_s$ representation in case of $n = 2$ processes.

Definition 3.8 (*Dijkstra's algorithm*) Components of the array b are represented by processes $\mathsf{B}_i(f)$ and $\mathsf{B}_i(t)$ $(i = 1,2)$ in Def. 3.5. The other shared variables are defined similarly. Let $i = 1,2$, $b_i, c_i \in \mathbb{B}$, and $k \in \mathbb{K}$; as usual, $\mathsf{PV}(b_1,b_2,c_1,c_2,k)$ denotes the parallel composition of all program variables. Its definition is as expected and, hence, omitted. Processes $\mathtt{P}_1$ and $\mathtt{P}_2$ are instead given below:

$$
\begin{array}{ll}
P_1 = \mathtt{req}_1.b_1wf.P_{11} + \tau.P_1 & P_2 = \mathtt{req}_2.b_2wf.P_{21} + \tau.P_2 \\
P_{11} = kr1.P_{15} + kr2.c_1wt.P_{12} & P_{21} = kr2.P_{25} + kr1.c_2wt.P_{22} \\
P_{12} = get.(kr1.P_{13} + kr2.P_{14}) & P_{22} = get.(kr2.P_{23} + kr1.P_{24}) \\
P_{13} = b_1rt.put.kw1.P_{11} + b_1rf.put.P_{11} & P_{23} = b_2rt.put.kw2.P_{21} + b_2rf.put.P_{21} \\
P_{14} = b_2rt.put.kw1.P_{11} + b_2rf.put.P_{11} & P_{24} = b_1rt.put.kw2.P_{21} + b_1rf.put.P_{21} \\
P_{15} = c_1wf.(c_2rf.P_{11} + c_2rt.P_{16}) & P_{25} = c_2wf.(c_1rf.P_{21} + c_1rt.P_{26}) \\
P_{16} = \mathsf{cs}_1.c_1wt.b_1wt.P_1 & P_{26} = \mathsf{cs}_2.c_2wt.b_2wt.P_2
\end{array}
$$

Dijkstra's algorithm is defined as $\mathsf{Dijkstra} = (((P_1 \parallel_{\emptyset} P_2) \parallel_{\{get,put\}} \mathsf{BK}) \parallel_B \mathsf{PV}(t,t,t,t,1))/(B \cup \{get,put\})$ where $\mathsf{BK} = get.put.\mathsf{BK}$.

As in [13] we must ensure that whenever, during the execution of the statement "**if** $b[k]$ **then** $k \leftarrow i$", process $\mathtt{P}_i$ has read variable k but not yet $b[k]$, the other process cannot change the value of the former variable. Note that BK locks the variable k in writing mode when evaluating $b[k]$. Indeed, after a *get*-action, k can be written only after a subsequent *put*-action, i.e. once $b[k]$ has been read.

As other papers in the literature (see e.g. [1]), we cannot prove the liveness of the algorithm[3]. In case k is 1, process $\mathtt{P}_1$ can immediately enter its critical section (after setting $b[1]$ to false, both conditions $k \neq 1$ and $\neg c[2]$ are false), while process $\mathtt{P}_2$ has to wait until $b[1]$ becomes true (i.e. until $\mathtt{P}_1$ ends its critical section) and, hence, it can change k. If $\mathtt{P}_1$ is fast enough to perform its critical section, reset variables $c[1]$ and $b[1]$, and submit a further request (again, by setting $b[1]$ to false) before $\mathtt{P}_2$ can actually read $b[1]$, the latter process can never enter its critical section. This scenario is fair and, hence, admissible; see e.g. in [1] where Dijkstra is analysed by exploiting the model checker SMV ([1] and references therein). The fairness notion assumed in [1] ensures that each process executes infinitely often and that no process can stay in its critical or non-critical section forever. The next example shows that the above scenario is also admissible if one assumes fairness of actions and introduces reasonable non-blocking behaviours.

Example 3.9 Let us consider the following timed computation:

[3]Paper [1] studies the liveness of the same algorithms we consider here except for Lamport. In [1] it has been proven that Peterson and Knuth are live, but Dijkstra is not.

$$
\begin{array}{lcll}
D & = & ((P_1 \parallel_{\emptyset} P_2) \parallel_{\{get,put\}} \mathsf{BK}) \parallel_B \mathsf{PV}(t,t,t,t,1) & \xrightarrow{\texttt{req}_1\ b_1wf\ kr1} \\
 & & ((P_{15} \parallel_{\emptyset} P_2) \parallel_{\{get,put\}} \mathsf{BK}) \parallel_B \mathsf{PV}(f,t,t,t,1) & \xrightarrow{c_1wf\ c_2rt} \\
 & & ((P_{16} \parallel_{\emptyset} P_2) \parallel_{\{get,put\}} \mathsf{BK}) \parallel_B \mathsf{PV}(f,t,f,t,1) & \xrightarrow{\texttt{req}_2\ b_2wf} \\
 & & ((P_{16} \parallel_{\emptyset} P_{21}) \parallel_{\{get,put\}} \mathsf{BK}) \parallel_B \mathsf{PV}(f,f,f,t,1) & \xrightarrow{kr1\ c_2wt} \\
 & & ((P_{16} \parallel_{\emptyset} P_{22}) \parallel_{\{get,put\}} \mathsf{BK}) \parallel_B \mathsf{PV}(f,f,f,t,1) & \xrightarrow{get\ kr1} \\
R & = & ((P_{16} \parallel_{\emptyset} P_{24}) \parallel_{\{get,put\}} put.\mathsf{BK}) \parallel_B \mathsf{PV}(f,f,f,t,1) & \xrightarrow{1} \\
\underline{R} & = & ((\underline{P_{16}} \parallel_{\emptyset} \underline{P_{24}}) \parallel_{\{get,put\}} \underline{put}.\mathsf{BK}) \parallel_B \underline{\mathsf{PV}(f,f,f,t,1)} & \xrightarrow{\texttt{cs}_1\ c_1wt\ b_1wt} \\
 & & ((P_1 \parallel_{\emptyset} \underline{P_{24}}) \parallel_{\{get,put\}} \underline{put}.\mathsf{BK}) \parallel_B \mathsf{PV}(t,\underline{f},t,\underline{t},\underline{1}) & \xrightarrow{\texttt{req}_1\ b_1wf\ kr1} \\
 & & ((P_{15} \parallel_{\emptyset} \underline{P_{24}}) \parallel_{\{get,put\}} \underline{put}.\mathsf{BK}) \parallel_B \mathsf{PV}(f,\underline{f},t,\underline{t},\underline{1}) & \xrightarrow{c_1wf\ c_2rt} \\
Q & = & ((P_{16} \parallel_{\emptyset} \underline{P_{24}}) \parallel_{\{get,put\}} \underline{put}.\mathsf{BK}) \parallel_B \mathsf{PV}(f,\underline{f},f,\underline{t},\underline{1}) & \xrightarrow{1} \quad \underline{R}
\end{array}
$$

Along the cycle $\underline{R}/B \xrightarrow{\texttt{cs}_1\ \tau^2\ \texttt{req}_1\ \tau^4} Q/B \xrightarrow{1} \underline{R}/B$, the process $\texttt{P}_1$ repeatedly changes the value of b_1 from *false* to *true* and vice versa. As in Example 3.7, this means that it can block forever process $\texttt{P}_2$.

Proposition 3.10 Dijkstra *is not live under the assumption of fairness of actions.*

Knuth's algorithm There are two processes $\texttt{P}_1$ and $\texttt{P}_2$, two variables c_1 and c_2 that take values in $\{0,1,2\}$ and whose initial value is 0, and a turn variable k that takes values in $\{1,2\}$ and whose initial value is arbitrary. Process P_i ($i = 1,2$) is described as follows, where j is the index of the other process:

```
KNUTH()
while true
   do ⟨non-critical section⟩;
      c_i ← 1;
      if k = i then goto 6;
      if c_j ≠ 0 then goto 4;
      c_i ← 2;
      if c_j = 2 then goto 3;
      k ← i;
      ⟨critical section⟩;
      k ← j;
      c_i ← 0;
```

Definition 3.11 (*Knuth's algorithm*) The turn variable k is given in Def. 3.1 and modelled according to Def. 3.8. Variables c_1 and c_2 are represented as follows, where $i = 1,2$:

$$
\begin{array}{lcl}
\mathsf{C}_i(0) & = & \{c_ir0, c_iw0\} \triangleright (c_iw1.\mathsf{C}_i(1) + c_iw2.\mathsf{C}_i(2)) \\
\mathsf{C}_i(1) & = & \{c_ir1, c_iw1\} \triangleright (c_iw0.\mathsf{C}_i(0) + c_iw2.\mathsf{C}_i(2)) \\
\mathsf{C}_i(2) & = & \{c_ir2, c_iw2\} \triangleright (c_iw0.\mathsf{C}_i(0) + c_iw1.\mathsf{C}_i(1))
\end{array}
$$

Let $c_1, c_2 \in \{0,1,2\}$ and $k \in \mathbb{K}$. We let $\mathsf{PV}(c_1,c_2,k)$ to be the parallel composition of all program variables. Moreover, processes $\texttt{P}_1$ and $\texttt{P}_2$ are defined as follows:

$$
\begin{array}{ll}
P_1 = \texttt{req}_1.c_1w1.P_{11} + \tau.P_1 & P_2 = \texttt{req}_2.c_2w1.P_{21} + \tau.P_2 \\
P_{11} = kr1.P_{13} + kr2.P_{12} & P_{21} = kr2.P_{23} + kr1.P_{22} \\
P_{12} = c_2r0.P_{13} + c_2r1.P_{11} + c_2r2.P_{11} & P_{22} = c_1r0.P_{23} + c_1r1.P_{21} + c_1r2.P_{21} \\
P_{13} = c_1w2.P_{14} & P_{23} = c_2w2.P_{24}
\end{array}
$$

$$\begin{array}{ll} P_{14} = c_2r0.P_{15} + c_2r1.P_{15} + c_2r2.P_{16} & P_{24} = c_1r0.P_{25} + c_1r1.P_{25} + c_1r2.P_{26} \\ P_{15} = kw1.\mathsf{cs}_1.kw2.c_1w0.P_1 & P_{25} = kw2.\mathsf{cs}_2.kw1.c_2w0.P_2 \\ P_{16} = c_1w1.P_{11} & P_{26} = c_2w1.P_{21} \end{array}$$

We define $\mathsf{Knuth} = ((P_1 \parallel_\emptyset P_2) \parallel_B \mathsf{PV}(0,0,1))/B$.

We now provide an example that shows the existence of a catastrophic cycle in the reduced transition system of the modified Knuth. This example also implies Prop. 3.13.

Example 3.12 Let us consider the following timed computation:

$$\begin{array}{lcll} K & = & (P_1 \parallel_\emptyset P_2) \parallel_B \mathsf{PV}(0,0,1) & \xrightarrow{\mathtt{req}_2\, c_2w1} \\ & & (P_1 \parallel_\emptyset P_{21}) \parallel_B \mathsf{PV}(0,1,1) & \xrightarrow{kr1\, c_1r0\, c_2w2} \\ & & (P_1 \parallel_\emptyset P_{24}) \parallel_B \mathsf{PV}(0,2,1) & \xrightarrow{\mathtt{req}_1\, c_1w1} \\ & & (P_{11} \parallel_\emptyset P_{24}) \parallel_B \mathsf{PV}(1,2,1) & \xrightarrow{kr1\, c_1w2} \\ R & = & (P_{14} \parallel_\emptyset P_{24}) \parallel_B \mathsf{PV}(2,2,1) & \xrightarrow{1} \\ \underline{R} & = & (\underline{P_{14}} \parallel_\emptyset \underline{P_{24}}) \parallel_B \underline{\mathsf{PV}(2,2,1)} & \xrightarrow{c_2r2} \\ & & (P_{16} \parallel_\emptyset \underline{P_{24}}) \parallel_B \underline{\mathsf{PV}(2,2,1)} & \xrightarrow{c_1w1\, kr1\, c_1w2} \\ Q & = & (P_{14} \parallel_\emptyset \underline{P_{24}}) \parallel_B \mathsf{PV}(2,\underline{2},\underline{1}) & \xrightarrow{1} \quad \underline{R} \end{array}$$

Once in R, process $\mathtt{P}_1$ cannot enter its critical section because c_2 is 2; but, the value of this variable will never change because $\mathtt{P}_2$ is blocked. Moreover, as in Examples 3.7 and 3.9, repeated changes of variable c_1 (from 2 to 1 and vice versa) allows a further 1-step in Q. The execution sequence $\mathsf{Knuth} = K/B \xrightarrow{\mathtt{req}_2\, \tau^4\, \mathtt{req}_2\, \tau^3} R/B \xrightarrow{\tau^4} R/B \ldots$ is fair but not live since process $\mathtt{P}_2$ never enters its critical section. Let us finally notice that Knuth is live e.g. in [1] since the above execution sequence is not fair as defined there, and hence not admissible, because process $\mathtt{P}_2$ does not execute infinitely often.

Proposition 3.13 Knuth *is not live under the assumption of fairness of actions.*

4 Related works and Conclusion

This work partly originates from [13] where Walker aimed at verifying six MUTEX algorithms with the Concurrency Workbench [4] (CWB, for short). Walker translated the algorithms into CCS and then verified the safety property that the two competing processes are never in their critical sections at the same time. Regarding the liveness property, Walker first considered the following interpretation – which could be expressed as a modal mu-calculus formula and then checked with the CWB:

> *An algorithm is live if whenever at some point in a computation the process* $\mathtt{P}_i$ *requests the execution of its critical section, then in any continuation 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) assumption 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 one 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. So, it may be no surprising that four of the six algorithms (Dekker, Dijkstra, Lamport and Hyman) fail to satisfy this property. Moreover, in order to economize on computational effort, the six algorithms in [13] have been minimized w.r.t. weak bisimulation. This allowed Walker to ignore some τ-loops that could invalidate the liveness property. And, indeed, all of them are not live whenever the formula expressing the first interpretation of liveness is evaluated over

	CWBNC	FASE		CWBNC	FASE		CWBNC	FASE
Dekker	✗	✓	*Peterson*	✗	✓	*Knuth*	✗	✗
Dijkstra	✗	✗	*Lamport*	✗	✗			

Table 3: Liveness of MUTEX solutions: CWBNC vs. FASE.

the transition system that does not abstract from τ's. By examining process $\mathtt{P}_i$, it is clear that these τ-loops arise, e.g. in Peterson, from repeated reading and writing of variables by the same process. This is common to all the algorithms and it is not introduced by the translation into CCS (or in PAFAS). Rather its presence reflects the faithfulness of the translation itself.

Then, Walker considered the same liveness property we study in Section 3. To establish that any of the algorithms is live under this second interpretation, Walker added some assumption. Indeed, one characteristic of the τ-loops arising from repeated reading and writing of variables by one process is that the other one is excluded from an infinite computation of the system. It is natural to ask if *only* the presence of such 'unfair' loops prevents any of the algorithms from being live. So, Walker proposed to use enriched formulas of the form $F \Rightarrow P$ where P is the property of interest (i.e. liveness) and F is a fairness assumption that assumes as admissible only those paths to which each process contributes infinitely often. Even if at the time of writing no automated analysis was possible, Walker discussed how fairness could be assumed. The basic idea is to tag each action with a unique *probe* or label; then, we can say the i-th process $\mathtt{P}_i$ contributes infinitely often to a computation whenever none of its probes is continuously possible from a certain point on. Finally, the liveness under this fairness assumption is expressed by letting K_i be the set of all probes of $\mathtt{P}_i$ and defining $FairLive = FairLive_1 \wedge FairLive_2$ where $FairLive_i = (\bigwedge_{a \in K_i} GF[a]false) \Rightarrow G(\langle \mathtt{req}_i \rangle true \Rightarrow F \langle \mathtt{cs}_i \rangle true)$, and the operators G (always), F (future), $\langle\rangle$ (possibly) and $[]$ (necessarily) are standard modal logics operators.

This fairness induced with probes is closely related to fairness of actions as it has been defined in [9, 10]. W.r.t. our characterisation (cf. Section 2), the main difference is that, instead of time and time passing, probes are used to decide whenever an action is continuously enabled along a computation and, hence, must be performed eventually. To allow a comparison, we have implemented these ideas within the *Concurrency Workbench of the New Century* [3] (CWBNC, for short) that, unlike CWB, can handle modal formulas with fairness constrains. To be able to attach a probe to each process action, the algorithms have been translated into *Timed CCS* (this is not possible by using the standard CCS language); probes are introduced by annotating synchronisation actions or τ's. For instance, the i-th processes of Peterson can be defined as follows:

$$P_i = b_iwt(req_i).kwj(a_i).P_{i1} \qquad P_{i2} = krj(a_i).P_{i1} + kri(a_i).P_{i3}$$
$$P_{i1} = b_jrf(b_i).P_{i3} + b_jrt(b_i).P_{i2} \qquad P_{i3} = \mathtt{cs}_i(cs_i).b_iwf(a_i).P_i$$

Note that two consecutive actions (as, e.g., b_iwt and kwj in P_i) never have the same label. Moreover, since the overall number of labels impacts on the computational effort (see below), we also try to reduce the number of labels we use. For example, we can reuse a_i to label the actions of P_{i2} because none of its actions is adiacent to kwj and this action has already been executed once P_{i2} is reached.

Whenever an action is performed, the corresponding label becomes *visible*[4] and can be used as a probe in *FairLive*. Table 3 shows that all the algorithms we consider are not live according to this second liveness interpretation (also in this setting, Lamport is live for process $\mathtt{P}_1$ but not for $\mathtt{P}_2$). As an example, consider a path from Peterson along which the first process reaches $P_{11} = b_2rf(b_1).P_{13} + b_2rt(b_1).P_{12}$,

[4]E.g., if P_i synchronises with $\mathsf{B}_i(f)$ on the execution of b_iwt, the label $@req_i$ becomes visible; similarly, whenever process P_i executes $\mathtt{cs}_i$ we get the label $@cs_i$.

	CWBNC	FASE		CWBNC	FASE		CWBNC	FASE
Dekker	103125	119	*Peterson*	4844	34	*Knuth*	110391	166
Dijkstra	110797	647	*Lamport*	1734	22			

Table 4: Execution time (expressed in milliseconds): CWBNC vs. FASE

b_2 is true and k is 2. Once in such a state, process $\mathtt{P}_1$ can read b_2 and k and come back to P_{11}. Along this cycle, no probe of $\mathtt{P}_1$ is continuously possible (probes b_1 and a_1 are alternately possible) but $\mathtt{cs}_1$ will never be performed. So, $FairLive_1$ is false and Peterson is not live. As in Example 3.4, there is a path along which a process can be indefinitely blocked by repeated reading. Also in this setting, the liveness of the algorithm strongly depends on the liveness of the hardware, i.e. on the the possibility of making some behaviours non-blocking.

As a further counter-check, we again consider Peterson but now we tag its actions in such a way that the same probe is associated to all the actions that appear along consecutive reading (trying to simulate the intuition behind non-blocking behaviours). So, let us replace P_{i2} with $P_{i2} = krj(b_i).P_{i1} + kri(b_i).P_{i3}$. Now, whenever in P_{11} and assuming b_2 and k equal to true and 2, the process $\mathtt{P}_1$ can still repeatedly read variables b_2 and k, but the corresponding path is not fair because probe b_1 is continuously possible. With these probes, Peterson and Dekker turn out to be live. So, probes can be used to somehow simulate non-blocking actions. But they must be added and (whenever necessary) tuned by the user by hand. This task is subject to errors and wrong assumptions that would give erroneous results. On the contrary, FASE can be more easily used by also a non-expert user that has only to decide whether (and, in case, which) non-blocking behaviours are necessary. In our opinion, the use of probes requires a deeper knowledge of the problem and much more attention in both modelling and analysis phases.

Another difference between the two approaches deals with *performance* issues. In Table 4 we report the execution time of both FASE and CWB-NC to perform the analysis on the algorithms discussed in this paper. In particular, in [2] an efficient algorithm for detecting catastrophic cycles has been proposed and implemented. This works in time $O(N+E)$ where N and E are, resp., the number of nodes and edges of the state space of the process. On the contrary, CWBNC uses an on-the-fly model checking algorithm whose complexity is exponential in the size of the formula (see [3]); in our case, this size strongly depends on the number of probes.

FASE is a good first step towards the creation of an integrated framework for the analysis of concurrent systems. The improvements introduced by the tool (and, in particular, the possibility to easily check non-functional properties such as liveness) allows us to derive results – as those in this paper – very hard to prove by hand. Since these results are very promising, we are currently planning to extend FASE in order to improve the analysis of more complex systems with a larger state space.

References

[1] N. Bogunovic & E. Pek (2003): *Verification of mutual exclusion algorithms with SMV system. EUROCON 2003. Computer as a Tool. The IEEE Region 8* 2(3), pp. 21–25, doi:10.1109/EURCON.2003.1248127. Available at http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1248127.

[2] F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini & W. Vogler (2009): *Evaluating the Efficiency of Asynchronous Systems with FASE*. In: *pre-proceedings of the 1st Int. Workshop on Quantitative Formal Methods*, Eindhoven (The Netherlands), pp. 101–106. Available at http://arxiv.org/abs/1105.1369.

[3] R. Cleaveland, T. Li & S. Sims (2000): *The Concurrency Workbench of the New Century User's Manual*. SUNY, Stony Brook, NY 11794-4400. Available at `http://www.simsnet.org/papers/cwb-user.pdf`.

[4] R. Cleaveland, J. Parrow & B. Steffen (1989): *The Concurrency Workbench.* In Joseph Sifakis, editor: *Automatic Verification Methods for Finite State Systems*, *Lecture Notes in Computer Science* 407, Springer, pp. 24–37.

[5] F. Corradini, M.R. Di Berardini & W. Vogler (2006): *Checking a Mutex Algorithm in a Process Algebra with Fairness.* In Christel Baier & Holger Hermanns, editors: *CONCUR*, *Lecture Notes in Computer Science* 4137, Springer, pp. 142–157, doi:10.1007/11817949_10. Available at `http://www.springerlink.com/content/jl71201p7585m448/`.

[6] F. Corradini, M.R. Di Berardini & W. Vogler (2009): *Time and Fairness in a Process Algebra with Non-blocking Reading.* In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma & Frank D. Valencia, editors: *SOFSEM*, *Lecture Notes in Computer Science* 5404, Springer, pp. 193–204, doi:10.1007/978-3-540-95891-8_20. Available at `http://www.springerlink.com/content/56vq063482528637/`.

[7] F. Corradini & W. Vogler (2005): *Measuring the performance of asynchronous systems with PAFAS*. *Theor. Comput. Sci.* 335(2-3), pp. 187–213, doi:10.1016/j.tcs.2004.01.039. Available at `http://portal.acm.org/citation.cfm?id=1085671`.

[8] F. Corradini, W. Vogler & L. Jenner (2002): *Comparing the worst-case efficiency of asynchronous systems with PAFAS*. *Acta Informatica* 38(11), pp. 735–792, doi:10.1007/s00236-002-0094-3. Available at `http://www.springerlink.com/content/2cuweb33bt0c1yqy/`.

[9] G. Costa & C. Stirling (1984): *A Fair Calculus of Communicating Systems.* *Acta Inf.* 21, pp. 417–441, doi:10.1007/BF00271640. Available at `http://www.springerlink.com/content/w7lh13t23n670858/`.

[10] G. Costa & C. Stirling (1987): *Weak and Strong Fairness in CCS*. *Inf. Comput.* 73(3), pp. 207–244, doi:10.1016/0890-5401(87)90013-7. Available at `http://portal.acm.org/citation.cfm?id=29965`.

[11] R. Milner (1989): *Communication and concurrency*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[12] R. De Nicola & M. Hennessy (1984): *Testing equivalences for processes*. *TCS* 34, pp. 83–133, doi:10.1016/0304-3975(84)90113-0. Available at `http://www.springerlink.com/content/m5110675433345x7/`.

[13] D. J. Walker (1989): *Automated Analysis of Mutual Exclusion Algorithms using CCS.* *Formal Asp. Comput.* 1(3), pp. 273–292, doi:10.1007/BF01887209. Available at `http://www.springerlink.com/content/m80614vt26566802/`.