Analysing Mutual Exclusion using Process Algebra with Signals

Citation for published version:
Dyseryn, V, van Glabbeek, R & Höfner, P 2017, Analysing Mutual Exclusion using Process Algebra with Signals. in K Peters & S Tini (eds), *Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics.* vol. 255, Electronic Proceedings in Theoretical Computer Science, vol. 255, Open Publishing Association, pp. 18-34, Combined 24th International Workshop on Expressiveness in Concurrency and the 14th Workshop on Structural Operational Semantics, Berlin, Berlin, Germany, 4/09/17. https://doi.org/10.4204/EPTCS.255.2

Digital Object Identifier (DOI):
10.4204/EPTCS.255.2

Published In:
Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics

Analysing Mutual Exclusion using Process Algebra with Signals

Victor Dyseryn
Ecole Polytechnique, Paris, France
Data61, CSIRO, Sydney, Australia
victor.dyseryn-fostier@polytechnique.edu

Rob van Glabbeek & Peter Höfner
Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
rvg@cs.stanford.edu Peter.Hoefner@data61.csiro.au

In contrast to common belief, the Calculus of Communicating Systems (CCS) and similar process algebras lack the expressive power to accurately capture mutual exclusion protocols without enriching the language with fairness assumptions. Adding a fairness assumption to implement a mutual exclusion protocol seems counter-intuitive. We employ a signalling operator, which can be combined with CCS, or other process calculi, and show that this minimal extension is expressive enough to model mutual exclusion: we confirm the correctness of Peterson's mutual exclusion algorithm for two processes, as well as Lamport's bakery algorithm, under reasonable assumptions on the underlying memory model. The correctness of Peterson's algorithm for more than two processes requires stronger, less realistic assumptions on the underlying memory model.

1 Introduction

In the process algebra community it is common belief that, on some level of abstraction, any distributed system can be modelled in standard process-algebraic specification formalisms like the Calculus of Communicating Systems (CCS) [26].

However, this sentiment has been proven incorrect [20]: two of the authors presented a simple fair scheduler—one that in suitable variations occurs in many distributed systems—of which *no* implementation can be expressed in CCS, unless CCS is enriched with a fairness assumption. Instances of our fair scheduler, that hence cannot be rendered correctly, are the *First in First out*[1], *Round Robin*, and *Fair Queueing* scheduling algorithms[2] as used in network routers [28, 29] and operating systems [23], or the *Completely Fair Scheduler*[3], which is the default scheduler of the Linux kernel since version 2.6.23. Since fair schedulers can be implemented in terms of mutual exclusion, this result implies that mutual exclusion protocols, such as the ones by Dekker [13, 15], Peterson [31] and Lamport [25], cannot be rendered correctly in CCS without imposing a fairness assumption.

Close approximations of Dekker's and Peterson's protocols rendered in CCS or similar formalisms abound in the literature [34, 5, 32, 16, 2]. Unless one makes a fairness assumption these renderings do not possess the liveness property that when a process leaves its non-critical section, and thus wants to enter the critical section, it will eventually succeed in doing so. When assuming fairness, this problem disappears [9]. However, since mutual exclusion protocols are often employed to ensure that each of several tasks gets allocated a fair amount of a shared resource, assuming fairness to implement mutual exclusion appears counter-intuitive.

Informally speaking, the reason why the CCS rendering of algorithms such as Peterson's does not work, is that it is possible that a process never gets a chance to write to a shared variable to indicate interest in entering the critical section. This is because other processes running in parallel and competing

[1] Also known as First Come First Served (FCFS)
[2] http://en.wikipedia.org/wiki/Scheduling_(computing)
[3] http://en.wikipedia.org/wiki/Completely_Fair_Scheduler

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 18–34, doi:10.4204/EPTCS.255.2

$$\alpha.P \xrightarrow{\alpha} P \qquad \frac{P_j \xrightarrow{\alpha} P'}{\sum_{i\in I} P_i \xrightarrow{\alpha} P'}\ (j \in I)$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{P \xrightarrow{a} P',\ Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ (\alpha, \bar{\alpha} \notin L) \qquad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ (A \stackrel{def}{=} P)$$

Table 1: Structural operational semantics of CCS

for the critical section are 'too busy' reading the shared variable all the time.

In this paper we extend CCS with *signals*. This extension is able to express mutual exclusion protocols *without* the use of fairness assumptions. To prove correctness, one only needs basic assumptions such as progress and justness.

We will use this extension to analyse the correctness of some of the most famous protocols for mutual exclusion, namely Peterson's algorithm, the filter lock algorithm—Peterson's algorithm for more than two processes—and Lamport's bakery algorithm. With regards to the filter lock algorithm our analysis reveals some surprising protocol behaviour.

2 Preliminaries: The Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) [26] is a process algebra, which is used to describe concurrent processes.

It is parametrised with sets $\mathscr{A}$ and $\mathscr{K}$ of *names* and *agent identifiers*. We define the set of *handshake actions* as $\mathscr{H} := \mathscr{A} \uplus \bar{\mathscr{A}}$, where $\bar{\mathscr{A}} := \{\bar{a} \mid a \in \mathscr{A}\}$ is the set of *co-names*. Complementation is extended to $\mathscr{H}$ by setting $\bar{\bar{a}} = a$. Finally, $Act := \mathscr{H} \uplus \{\tau\}$ is the set of *actions*, where τ is a special *internal action*. In this paper $a, b, c, \ldots$ range over $\mathscr{H}$, α, β over Act, and A, B range over $\mathscr{K}$. A *relabelling* is a function $f : \mathscr{H} \to \mathscr{H}$ satisfying $f(\bar{a}) = \overline{f(a)}$; it extends to Act by $f(\tau) := \tau$. Each $A \in \mathscr{K}$ comes with a defining equation $A \stackrel{def}{=} P$ with P being a CCS expression as defined below.

The class $\mathscr{T}_{\mathrm{CCS}}$ of *CCS expressions* is defined as the smallest class that includes

- *agent identifiers* $A \in \mathscr{K}$;
- *prefixes* $\alpha.P$;
- *(infinite) choices* $\sum_{i\in I} P_i$;
- *parallel compositions* $P|Q$;
- *restrictions* $P\backslash L$;
- *relabellings* $P[f]$;

where $P, P_i, Q \in \mathscr{T}_{\mathrm{CCS}}$ are CCS expressions, I an index set, $L \subseteq \mathscr{A}$ a set of names, and f an arbitrary relabelling function. In case $I = \{1,2\}$, we write $P_1 + P_2$ for $\sum_{i\in I} P_i$. The *inactive process* $\mathbf{0}$ is defined by $\sum_{i\in\emptyset} P_i$; it is not capable to perform any action.

The semantics of CCS is given by the labelled transition relation $\to \subseteq \mathscr{T}_{\mathrm{CCS}} \times Act \times \mathscr{T}_{\mathrm{CCS}}$, where transitions $P \xrightarrow{\alpha} Q$ are derived from the rules of Table 1. The process $\alpha.P$ performs the action α first and subsequently acts as P. The choice operator $\sum_{i\in I} P_i$ may act as any of the P_i, depending on which of the processes is able to act at all. The parallel composition $P|Q$ executes an action from P, an action from Q, or in the case where P and Q can perform complementary actions a and $\bar{a}$, the process can perform a synchronisation, resulting in an internal action τ. The restriction operator $P\backslash L$ inhibits execution of the actions from L and their complements. The only way for a subprocess of $P\backslash L$ to perform an action $a \in L$ is through synchronisation with another subprocess of $P\backslash L$, which performs $\bar{a}$. The relabelling $P[f]$ acts

like process P with all labels a replaced by $f(a)$. Last, the rule for agent identifiers says that an agent A has the same transitions as the body P of its defining equation.

As usual, to avoid parentheses, we assume that the operators have decreasing binding strength in the following order: restriction and relabelling, prefixing, parallel composition, choice.

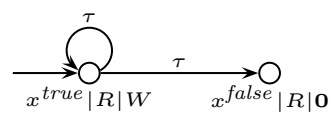
The pair $\langle \mathscr{T}_{\mathrm{CCS}}, \rightarrow \rangle$ is called the *labelled transition system* (LTS) of CCS.

Example 1 We describe a simple shared memory system in CCS, using the name $asgn_x^v$ for the assignment of value v to the variable x, and n_x^v for noticing or notifying that the variable x has the value v. The action $\overline{asgn_x^v}$ communicates the assignment $x := v$ to the shared memory, whereas $asgn_x^v$ is the action of the shared memory of accepting this communication. Likewise, $\overline{n_x^v}$ is a notification by the shared memory that x equals v; it synchronises with the complementary action n_x^v of noticing that $x = v$.

We consider the process $(x^{true} \,|\, R \,|\, W)\backslash\{asgn_x^{true}, asgn_x^{false}, n_x^{true}, n_x^{false}\}$, where

$$x^{true} \stackrel{def}{=} asgn_x^{true} \,.\, x^{true} \;+\; asgn_x^{false} \,.\, x^{false} \;+\; \overline{n_x^{true}} \,.\, x^{true} \,,$$
$$x^{false} \stackrel{def}{=} asgn_x^{true} \,.\, x^{true} \;+\; asgn_x^{false} \,.\, x^{false} \;+\; \overline{n_x^{false}} \,.\, x^{false} \,,$$
$$R \stackrel{def}{=} n_x^{true} \,.\, R \qquad \text{and} \qquad W \stackrel{def}{=} \overline{asgn_x^{false}} \,.\, \mathbf{0} \,.$$

The processes x^{true} and x^{false} model the two states of a shared Boolean variable x (*true* and *false*, respectively). Both accept assignment actions, changing their state accordingly. They also provide their respective value to a potential reader. The process R (*reader*) is an infinite loop which permanently tries to read value *true* from variable x, and the process W (*writer*) tries once to set variable x to false. Since the overall process uses the restriction operator, its transition system, depicted on the right, has only two transitions, a τ-loop of R reading the value, and a transition to $x^{false}|R|\mathbf{0}$ of W assigning x to false—after that no further transition is possible. The justness assumption, to be described in Sects. 4 and 5, is not sufficient to ensure that the writer eventually performs its transition, and this fact is one of the motivations why we introduce signals in Sect. 6.



3 Peterson's Mutual Exclusion Protocol—Part I

In [20] it is shown that Peterson's mutual exclusion protocol [31] cannot be expressed in CCS without assuming fairness. In this section we briefly recapitulate the protocol itself and present an optimal rendering in CCS. In the next section we discuss what the problems are with such a rendering.

The 'classical' Peterson's mutual exclusion protocol deals with two concurrent processes A and B that want to alternate critical and noncritical sections.

Each of the processes will stay only a finite amount of time in the critical section, although it is allowed to stay forever in its noncritical section. The purpose of the algorithm is to ensure that the processes are never simultaneously in the critical section, and to guarantee that both processes keep making progress; in particular the latter means that if a process wants to access the critical section it will eventually do so.

A pseudocode rendering of Peterson's protocol is depicted in Fig. 1. The processes use three shared variables: *readyA*, *readyB* and *turn*. The Boolean variable *readyA* can be written by Process A and read by Process B, whereas *readyB* can be written by B and read by A. By setting *readyA* to *true*, Process A signals to Process B that it wants to enter the critical section. The variable *turn* can be written and read by both processes. Its carefully designed functionality guarantees mutual exclusion as well as deadlock-freedom. Both *readyA* and *readyB* are initialised with *false* and *turn* with *A*.

Process A

repeat forever

ℓ_1 **noncritical section**
ℓ_2 $readyA := true$
ℓ_3 $turn := B$
ℓ_4 **await** $(readyB = false \vee turn = A)$
ℓ_5 **critical section**
ℓ_6 $readyA := false$

Process B

repeat forever

m_1 **noncritical section**
m_2 $readyB := true$
m_3 $turn := A$
m_4 **await** $(readyA = false \vee turn = B)$
m_5 **critical section**
m_6 $readyB := false$

Figure 1: Peterson's algorithm (pseudocode)

In order to model this protocol in CCS, we use the names **noncritA**, **critA**, **noncritB**, and **critB**, for Processes A and B executing their (non)critical section. The names $asgn^v_x$ and n^v_x for the interactions of A and B with a shared memory have been defined in Ex. 1. The Processes A and B can be modelled as

$$\mathrm{A} \stackrel{def}{=} \mathbf{noncritA} \,.\, \overline{asgn^{true}_{readyA}} \,.\, \overline{asgn^{\mathrm{B}}_{turn}} \,.\, (n^{false}_{readyB} \;+\; n^{\mathrm{A}}_{turn}) \,.\, \mathbf{critA} \,.\, \overline{asgn^{false}_{readyA}} \,.\, \mathrm{A}\,,$$
$$\mathrm{B} \stackrel{def}{=} \mathbf{noncritB} \,.\, \overline{asgn^{true}_{readyB}} \,.\, \overline{asgn^{\mathrm{A}}_{turn}} \,.\, (n^{false}_{readyA} \;+\; n^{\mathrm{B}}_{turn}) \,.\, \mathbf{critB} \,.\, \overline{asgn^{false}_{readyB}} \,.\, \mathrm{B}\,,$$

where $(a+b).P$ is a shorthand for $a.P + b.P$. This CCS rendering naturally captures the **await** statement, requiring Process A to wait at instruction ℓ_4 until it can read that $readyB = false$ or $turn = A$. We use two agent identifiers for each Boolean variable, one for each value, similar to Ex. 1. For example, we have $Turn^{\mathrm{A}} \stackrel{def}{=} asgn^{\mathrm{A}}_{turn} \,.\, Turn^{\mathrm{A}} + asgn^{\mathrm{B}}_{turn} \,.\, Turn^{\mathrm{B}} \;+\; \overline{n^{\mathrm{A}}_{turn}} \,.\, Turn^{\mathrm{A}}$. Peterson's algorithm is the parallel composition of all these processes, restricting all the communications

$$(\mathrm{A} \,|\, \mathrm{B} \,|\, ReadyA^{false} \,|\, ReadyB^{false} \,|\, Turn^{A}) \backslash L\,,$$

where L is the set of all names except **noncritA**, **critA**, **noncritB**, and **critB**.

It is well known that Peterson's protocol satisfies the safety property that both processes are never in the critical section at the same time. In terms of Fig. 1, there is no reachable state where A and B have already executed lines ℓ_4 and m_4 but have not yet executed ℓ_6 or m_6 [31, 34]. The validity of the liveness property, that any process leaving its noncritical section will eventually enter the critical section, depends on its precise formalisation, as discussed in Sects. 4 and 5.

4 Why the CCS Rendering of Peterson's Algorithm is Unsatisfactory

Liveness properties generally only hold under some assumptions. The intended liveness property for Peterson's algorithm may already be violated if both processes come to a permanent halt for no apparent reason. This behaviour should be considered unrealistic. To rule it out one usually makes a *progress* assumption, formalised in Sect. 5, which can be formulated as follows [17, 21]:

Any process in a state that admits a non-blocking action will eventually perform an action.

Another example is an execution path ρ in which first Process A completes instruction ℓ_1; leaving its noncritical section it implicitly wishes to enter the critical section. Subsequently, Process B cycles through its complete list of instructions in perpetuity without A making any further progress. This is possible because *readyA* is never updated and always evaluates to false. This execution path, if admitted, would be another counterexample to the intended liveness property. However, progress is not sufficient to rule out such a path; after all the whole system is making progress. To rule it out as a valid system run, we need the stronger assumption of *justness* [21], or an even stronger *fairness* assumption [18].

We formalise justness in the next section. Here we sketch the general idea, and the difference with fairness, by an example.

Suppose we have a vending machine with a single slot for inserting coins, and there are two customers: one intends to insert an infinite supply of quarters, and the other an infinite supply of dimes. No customer intends to ever extract something from the vending machine. Since a quarter and a dime cannot be entered simultaneously, the two customers compete for a shared resource. Should it be allowed for one customer to enter an unending sequence of quarters, while the other does not even get in a single dime? The assumption of (strong or weak) fairness rules out this realistic behaviour, while weaker assumptions like progress and justness allow this as one of the valid ways such interactions between the customers and the vending machine may play out.

Alternatively, assume that the same two customers have access to a vending machine each, and that each of these vending machines serves that customer only. In that case the assumption of progress is not strong enough to rule out that one customer enters an unending sequence of quarters, while the other does not even get in a single dime; after all the whole system keeps making progress at all times. The assumption of justness guarantees that the customer will get a chance to enter his dimes by applying the idea of progress to isolated components of a system; it entails that the perpetual insertion of quarters by one customer in one machine in no way prevents the other customer to insert dimes in the other machine.

In [34] Walker shows that once Process A executes instruction ℓ_2, it will in fact enter the critical section, i.e. execute ℓ_4. This proof assumes progress, but not justness, let alone fairness. The only question left is whether we can guarantee that execution of ℓ_1 is always followed by ℓ_2. Thus, when assuming progress, the only possible counterexample to the intended liveness property of Peterson's algorithm is the execution path ρ sketched above, and its symmetric counterpart. This execution represents a battle for the shared variable *readyA*. Process A tries to assign a value to this variable, whereas Process B engages in an unending sequence of read actions of this variable (as part of infinitely many instructions m_4). If we assume that the central memory in which variable *readyA* is stored implements its own mutual exclusion protocol, which prevents two processes from reading and writing the same variable at the same time (but guarantees no liveness property), we may have the situation that Process A has to wait before setting *readyA* to *true* until Process B is done reading this variable. However, Process B may be so quick that each time it is done reading *readyA* it executes m_5–m_3 in the blink of an eye and grabs hold of the same variable for reading it again before Process A gets a chance to write to it. Under this assumption we would conclude that Peterson's algorithm does not have the required liveness property, since Process A may never get a chance to write to *readyA* because Process B is too busy reading it, and hence never ever enters the critical section.

However, it is reasonable to assume that in the intended setting where Peterson's algorithm would be employed, the central memory does *not* employ its own mutual exclusion protocol that prevents one process from writing a variable while another is reading it.[4] With this view of the central memory in mind, instruction ℓ_2 cannot be blocked by Process B, and hence the assumption of justness is sufficient to ensure that Peterson's protocol *does* have the required liveness property.

The same conclusion cannot be drawn for the rendering of Peterson's algorithm in CCS. Here the write action $\ell_2 = \overline{asgn^{true}_{readyA}}$ needs to synchronise with the action $asgn^{true}_{readyA}$ of the shared memory storing variable *readyA*. That process has to make a choice between executing $asgn^{true}_{readyA}$ and executing $\overline{n^{false}_{readyA}}$, the latter in synchronisation with Process B . When it chooses $\overline{n^{false}_{readyA}}$ it has to wait until this instruction is terminated before the same choice arises again. Hence the write action can be blocked by the read action, and justness is not strong enough an assumption to ensure that eventually the assignment will

[4]Without such a protocol, it could be argued that the reading process may read *anything* when reading overlaps with writing the same variable. However, the variable *readyA* has only two possible values that can be read, and depending on which of these is returned, the overlapping read action may just as well be thought to occur before or after the write action.

take place. Assuming fairness is of course enough to achieve this, but risky since it has the potential to rule out realistic behaviour (see above).

The above reasoning merely shows that the given implementation of Peterson's mutual exclusion protocol in CCS requires fairness to be correct. In [20] we show that the same holds for any implementation of any mutual exclusion protocol in CCS, and the same argument applies to a wider class of process algebras. Peterson expressed his protocol in pseudocode without resorting to a fairness assumption. We understand that he assumes progress and justness implicitly, and accordingly his protocol and liveness claim are correct. It follows that Peterson's pseudocode does not admit an accurate translation into CCS.

5 Formalising Progress and Justness

Liveness properties are naturally expressed as properties of execution paths. A *path* of a process P is an alternating sequence $P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} P_n \xrightarrow{\alpha_{n+1}} \ldots$ of states and transitions. A path can be finite or infinite. A possible formulation of the liveness property of Peterson's algorithm, applied to paths π, is that each occurrence of a transition labelled with **noncritA** in π is followed by an occurrence of **critA**, and similarly for B. To express when a liveness property holds for a system P, we need the notion of a *complete* path: one that describes a complete execution of P, rather than a partial one. The property holds for P iff it holds for all its complete paths. Progress, justness and fairness assumptions rule out certain paths from being considered complete—those that are in disagreement with the assumption. The stronger the assumption, the more paths are ruled out, and the more likely it is that a given liveness property holds.

A complete path ending in a state P_n models a system run in which no further activity takes place after P_n has been reached. A complete path ending in a transition models a system run where transitions are considered to have a duration, and the final transition commenced, but never finishes.

One assumption we adopt in this paper is that "atomic actions always terminate" [30]. It rules out all paths ending in a transition. To check whether Peterson's algorithm is compatible with this assumption, we note that processes are not allowed to stay forever in their critical sections, so the actions **critA** and **critB** can be assumed to terminate. Read and write actions of variables terminate as well. However, a process is allowed to stay forever in its noncritical section, so the actions **noncritA** and **noncritB** need not terminate. To make our formalisation of Peterson's algorithm compatible with the assumption that actions terminate, we could split the action **noncritA** into **start**(**noncritA**) and **end**(**noncritA**). Both these actions terminate, and an execution in which Process A stays in its noncritical section corresponds with a complete path that ends in the state between these transitions. To save the effort of rewriting the protocol from Sect. 3, we shall identify instruction ℓ_6 with entering the noncritical section, and interpret ℓ_1 as leaving the noncritical section. Thus, the processes start out being in their noncritical sections.

To formalise the assumptions of progress and justness, we need the concept of a *non-blocking* action. A process of the form $\tau.P$ should surely execute the internal action τ, and not stay forever in its initial state. However, a process $a.P$ running in the environment $(\_ \mid E)\backslash\{a\}$ may very well stay in its initial state, namely when the environment E never provides a signal $\bar{a}$ that the process can read. With this in mind we assume a classification of the set of actions into blocking and non-blocking actions [21]. The internal action τ is always non-blocking, and any action a classified as non-blocking shall never be put in the scope of a restriction operator $\backslash L$ with $a \in L$, and never be renamed into a blocking action [20]. The transition system of Peterson's algorithm features actions **critA**, **critB**, **noncritA**, **noncritB** and τ—other names are forbidden by the restriction operator. We classify **critA**, **critB** and τ as non-blocking, but the actions **noncritA** and **noncritB** of leaving the noncritical sections may block. Our progress

assumption rules out as complete any path ending in a state in which a non-blocking action is enabled, i.e. any system state except for the ones where both Processes A and B are (back) in their initial state.

The (stronger) *justness assumption* from [21] is:

> *If a combination of components in a parallel composition is in a state that admits a non-blocking action, then one (or more) of them will eventually partake in an action.*

Its formalisation uses *decomposition*: a transition $P|Q \xrightarrow{\alpha} R$ derives, through the rules of Table 1, from

- a transition $P \xrightarrow{\alpha} P'$ and a state Q, where $R = P'|Q$,
- two transitions $P \xrightarrow{a} P'$ and $Q \xrightarrow{\bar{a}} Q'$, where $R = P'|Q'$ and $\alpha = \tau$,
- or from a state P and a transition $Q \xrightarrow{\alpha} Q'$, where $R = P|Q'$.

This transition/state, transition/transition or state/transition pair is called a *decomposition* of $P|Q \xrightarrow{\alpha} R$; it need not be unique. A *decomposition* of a path π of $P|Q$ into paths π_1 and π_2 of P and Q, respectively, is obtained by decomposing each transition in the path, and concatenating all left-projections into a path of P—the *decomposition of* π *along* P—and all right-projections into a path of Q. It could be that a path π is infinite, yet either π_1 or π_2 (but not both) are finite. Decomposition of paths need not be unique.

Similarly, any transition $P[f] \xrightarrow{\alpha} R$ stems from a transition $P \xrightarrow{\beta} P'$, where $R = P'[f]$ and $\alpha = f(\beta)$. This transition is called a decomposition of $P[f] \xrightarrow{\alpha} R$. A *decomposition* of a path π of $P[f]$ is obtained by decomposing each transition in the path, and concatenating all transitions so obtained into a path of P. A decomposition of a path of $P\backslash L$ is defined likewise.

Definition 1 The class of Y-*just* paths, for $Y \subseteq \mathscr{H}$, is the largest class of paths in $\mathscr{T}_{\text{CCS}}$ such that

- a finite Y-just path ends in a state that admits actions from Y only;
- a Y-just path of a process $P|Q$ can be decomposed into an X-just path of P and a Z-just path of Q such that $Y \supseteq X \cup Z$ and $X \cap \bar{Z} = \emptyset$—here $\bar{Z} := \{\bar{c} \mid c \in Z\}$;
- a Y-just path of $P\backslash L$ can be decomposed into a $Y \cup L \cup \bar{L}$-just path of P;
- a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
- and each suffix of a Y-just path is Y-just.

A path π is *just* if it is Y-just for some set of blocking actions $Y \subseteq \mathscr{H}$. A just path π is *a-enabled* for an action $a \in \mathscr{H}$ if $a \in Y$ for all Y such that π is Y-just.

Intuitively, a Y-just path models a run in which Y is an upper bound of the set of labels of abstract transitions[5] that from some point onwards are continuously enabled but never taken. Here an abstract transition with a label from $\mathscr{H}$ is deemed to be continuously enabled but never taken iff it is enabled in a parallel component that performs no further actions. Such a run can occur in the modelled system if the environment from some point onwards blocks the actions in Y.

Now consider the path ρ violating the intended liveness property. The decompositions of ρ along Processes A and B were mentioned in Sect. 4. These paths are $\{\overline{asgn^{true}_{readyA}}\}$-just and $\emptyset$-just, respectively. The decomposition along $Turn^A$ is an infinite path taking action $asgn^A_{turn}$ only ($\emptyset$-just). The decomposition along $ReadyB^{false}$ is an infinite path alternatingly taking actions $asgn^{true}_{readyB}$ and $asgn^{false}_{readyB}$ (also $\emptyset$-just) and the decomposition along $ReadyA^{false}$ is an infinite path taking action $\overline{n^{false}_{readyA}}$ only (again $\emptyset$-just). It follows that the composition ρ of these five paths is $\emptyset$-just. Intuitively this is the case because no communication is permanently enabled and never taken. In particular, the communication $\overline{asgn^{true}_{readyA}}$ is disabled each time the component $ReadyA^{false}$ does the action $\overline{n^{false}_{readyA}}$ instead.

[5] The CCS process $a.0|b.0$ has two transitions labelled a, namely $a.0|b.0 \xrightarrow{a} 0|b.0$ and $a.0|0 \xrightarrow{a} 0|0$. The only difference between these two transitions is that one occurs before the action b is performed by the parallel component and the other afterwards. In [21] we formalise a notion of an *abstract transition* that identifies these two concrete transitions.

$(P\hat{\ }s)\frown^{s}$	$\dfrac{P \xrightarrow{\alpha} P'}{P\hat{\ }s \xrightarrow{\alpha} P'}$	$\dfrac{P\frown^{s}}{(P\hat{\ }t)\frown^{s}}$	$\dfrac{P_j\frown^{s}}{(\sum_{i\in I} P_i)\frown^{s}}\ (j\in I)$
$\dfrac{P\frown^{s}}{(P\mid Q)\frown^{s}}$	$\dfrac{P\frown^{s},\ Q \xrightarrow{s} Q'}{P\mid Q \xrightarrow{\tau} P\mid Q'}$	$\dfrac{P \xrightarrow{s} P',\ Q\frown^{s}}{P\mid Q \xrightarrow{\tau} P'\mid Q}$	$\dfrac{Q\frown^{s}}{(P\mid Q)\frown^{s}}$
$\dfrac{P\frown^{s}}{(P\backslash L)\frown^{s}}\ (s\notin L)$	$\dfrac{P\frown^{s}}{P[f]\frown^{f(s)}}$	$\dfrac{P\frown^{s}}{A\frown^{s}}\ (A \stackrel{def}{=} P)$	

Table 2: Structural operational semantics for signals of CCSS

6 CCS with Signals

We would like to prevent such a path to be complete in a CCS model of Peterson's algorithm. In order to achieve this, we propose to replace an action such as $\overline{n^{false}_{readyB}}$, which makes the variable busy even if it is only read, by a state predicate providing its value. This mode of communication is called *signalling*.

CCS with signals (CCSS) is CCS extended with a signalling operator. Informally, the signalling operator $P\hat{\ }s$ emits the signal s to be read by another process. Signal emission cannot block other actions. Formally, CCS is extended with a set $\mathscr{S}$ of *signals*, ranged over by $s,t,\ldots$. In CCSS the set of actions is defined as $Act := \mathscr{S} \uplus \mathscr{H} \uplus \{\tau\}$. A relabelling is a function $f : (\mathscr{S} \to \mathscr{S}) \cup (\mathscr{H} \to \mathscr{H})$ satisfying $f(\bar{a}) = \overline{f(a)}$. As before it extends to Act by $f(\tau) = \tau$.

The class $\mathscr{T}_{\mathrm{CCSS}}$ of *CCSS expressions* is defined as the smallest class that includes

- *agent identifiers* $A \in \mathscr{K}$;
- *prefixes* $\alpha.P$;
- *(infinite) choices* $\sum_{i\in I} P_i$;
- *parallel compositions* $P|Q$;
- *restrictions* $P\backslash L$;
- *relabellings* $P[f]$;
- *signallings* $P\hat{\ }s$

where $P, P_i, Q \in \mathscr{T}_{\mathrm{CCSS}}$ are CCSS expressions, I an index set, $L \subseteq \mathscr{A} \cup \mathscr{S}$ a set of handshake names and signals, f an arbitrary relabelling function, and $s \in \mathscr{S}$ a signal. The new operator $\hat{\ }$ binds as strong as relabelling and restriction.

The semantics of CCSS is given by the labelled transition relation $\to \subseteq \mathscr{T}_{\mathrm{CCSS}} \times Act \times \mathscr{T}_{\mathrm{CCSS}}$ and a predicate $\frown \subseteq \mathscr{T}_{\mathrm{CCSS}} \times \mathscr{S}$ that are derived from the rules of CCS (Table 1, where α can also be a signal) and the new rules of Table 2. The predicate $P\frown^{s}$ indicates that process P emits the signal s, whereas a transition $P \xrightarrow{s} P'$ indicates that P reads the signal s and thereby turns into P'. The first rule is the base case showing that a process $P\hat{\ }s$ emits the signal s. The second rule models the fact that signalling cannot prevent a process from making progress. After having taken an action, the signalling process loses its ability to emit the signal. It is essentially this rule which fixes the read/write problem presented in the previous section. The two rules in the middle of Table 2 state that the action of reading a signal by one component in (parallel) composition together with the emission of the same signal by another component, results in an internal transition τ; similar to the case of handshake communication. Note that the component emitting the signal does not change through this interaction. All the other rules of Table 2 lift the emission of s by a subprocess P to the overall process. Table 2 can easily be adapted to other process calculi, hence our extension is not limited to CCS.

We give an example similar to the one at the end of Sect. 2 to illustrate the use of signals.

Example 2 We describe once again a one-variable shared memory system with an infinite reader R and a single writer W. But this time communication actions n^{v}_{x} and $\overline{n^{v}_{x}}$ are replaced with signals n^{v}_{x}. The variable x now emits a signal notifying its value, so we have: $x^{true} \stackrel{def}{=} (asgn^{true}_{x}\,.\,x^{true} + asgn^{false}_{x}\,.\,x^{false})\hat{\ }n^{true}_{x}$

and $x^{false} \stackrel{def}{=} (asgn_x^{true} \,.\, x^{true} + asgn_x^{false} \,.\, x^{false})\hat{}\, n_x^{false}$; the rest of the example remains unchanged. The transition system is exactly the same, but now justness guarantees that the variable x will eventually be set to *false*. This is in contrast to Ex. 1, where it is not guaranteed that x will eventually be *false*, even when assuming justness. More precisely, if only the reader takes actions, x^{true} is now not progressing because it is emitting a signal only, and then, assuming justness, it must eventually enter into communication with the writer.

As we have extended CCS with a novel operator, we have to make sure that our extension behaves 'naturally', in the way one would expect.

Theorem 1 *Strong bisimilarity [19] is a congruence for all operators of CCSS.*

Proof. We get the result directly from the existing theory on structural operational semantics, as a result of carefully designing our language. All rules of Tables 1 and 2 are in the *path* format of Baeten and Verhoef [3], and hence the theorem holds [3]. □

Theorem 2 *The operator | is associative and commutative, and the operator ˆ is pseudo-commutative, i.e. $P\hat{}s\hat{}t = P\hat{}t\hat{}s$, all up to bisimilarity.*

Proof. Our process algebra with predicates can easily be encoded in a process algebra without, by writing

$$P \xrightarrow{\bar{s}} P \text{ for } P^{\curvearrowright s} \,.$$

On the level of the structural operational semantics, this amounts to letting α range over $Act \cup \{\bar{s} \mid s \in \mathscr{S}\}$ in the rules of Table 1, and changing the first rule of Table 2 into $P\hat{}s \xrightarrow{\bar{s}} P\hat{}s$. The third rule of Table 2 becomes an instance of the second (with $\alpha \in Act \cup \{\bar{s} \mid s \in \mathscr{S}\}$), and the remaining rules of Table 2 become special cases of the rules of Table 1.

Clearly, two processes are bisimilar in the original CCSS iff they are bisimilar in this encoding. Since the parallel composition of the encoded CCSS is the same as the one of CCS, it is known to be associative and commutative up to bisimilarity [26].

To prove pseudo-commutativity of ˆ, we note that $P\hat{}s\hat{}t$ and $P\hat{}t\hat{}s$ have exactly the same outgoing transitions and signals, thereby being trivially equal up to bisimilarity. □

Since we extended CCS, we also have to extend our definition of justness. The decomposition of paths remains unchanged, except that a transition $P|Q \xrightarrow{\tau} R$ can now derive, through the rules of Table 2, from signal communication. In that case we consider the decomposition along the signalling process empty, just as if it was an application of the left- or right-parallel composition rule. Because processes can communicate through signalling, we first introduce the definition of signalling paths. Informally, a path emits signal s if one component in the parallel composition ends in a state where signal s is activated. A Y-signalling path is a path where Y is an upper bound on the signals emitted by the path.

Definition 2 The class of *Y-signalling* paths, for $Y \subseteq \mathscr{S}$, is the largest class of paths in $\mathscr{T}_{\text{CCS}}$ such that

- a finite Y-signalling path ends in a state that admits signals from Y only;
- a Y-signalling path of a process $P|Q$ can be decomposed into an X-signalling path of P and a Z-signalling path of Q such that $Y \supseteq X \cup Z$;
- a Y-signalling path of $P\backslash L$ can be decomposed into a $Y \cup L_{\mathscr{S}}$-signalling path of P—here $L_{\mathscr{S}} := L \cap \mathscr{S}$ restricts the set L to signals;
- a Y-signalling path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-signalling path of P;
- and each suffix of a Y-signalling path is Y-signalling.

Using this definition, we can adapt the definition of justness of Sect. 5.

Definition 3 The class of *Y-just* paths, for $Y \subseteq \mathscr{H} \cup \mathscr{S}$, is the largest class of paths in $\mathscr{T}_{\mathrm{CCSS}}$ such that

- a finite Y-just path ends in a state that admits actions from Y only;
- a Y-just path of a process $P|Q$ can be decomposed into a path of P that is X-just and X'-signalling, and a path of Q that is Z-just and Z'-signalling, such that $Y \supseteq X \cup Z$, $X \cap \bar{Z}_{\mathscr{H}} = \emptyset$, $X \cap Z' = \emptyset$ and $X' \cap Z = \emptyset$—here $\bar{Z}_{\mathscr{H}} := \{\bar{a} \mid a \in Z \cap \mathscr{H}\}$;
- a Y-just path of $P \backslash L$ can be decomposed into a $Y \cup L \cup \bar{L}_{\mathscr{H}}$-just path of P;
- a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
- and each suffix of a Y-just path is Y-just.

As before, a path π is *just* if it is Y-just for some set of blocking actions and signals $Y \subseteq \mathscr{H} \cup \mathscr{S}$. A just path π is *a-enabled* for $a \in \mathscr{H} \cup \mathscr{S}$ if $a \in Y$ for all Y such that π is Y-just.

The condition on signals in the second item guarantees that a process $(\mathbf{0}\hat{\ }s \,|\, s.\mathbf{0}) \backslash \{s\}$ makes progress.

The encoding in the proof of Theorem 2 does not preserve justness. In Ex. 2, for instance, applying the operational semantics of Table 2, the path ρ_R involving infinitely many read actions but no write action is not just, because its decomposition along x^{true} is finite and $asgn_x^{false}$-enabled, whereas its decomposition along W is $\overline{asgn_x^{false}}$-enabled; so by the second clause of Def. 3 ρ_R is not just: there are no Y, X and Z such that the condition $X \cap \bar{Z}_{\mathscr{H}} = \emptyset$ is satisfied. Yet, after applying the encoding in the proof of Theorem 2, the decomposition along x^{true} becomes infinite and $\emptyset$-just, and ρ_R becomes just. This is the main reason we did not present the semantics of CCSS in this form from the onset.

7 Peterson's Mutual Exclusion Protocol—Part II

We now present an implementation of Peterson's mutual exclusion algorithm in CCSS. We use the same notation as in Sect. 3, except that actions n_x^v and $\overline{n_x^v}$ are replaced with signals n_x^v, just as in Ex. 2. Only the variable processes change, such as $Turn^{\mathrm{A}} \stackrel{def}{=} (asgn_{turn}^{\mathrm{A}} . Turn^{\mathrm{A}} + asgn_{turn}^{\mathrm{B}} . Turn^{\mathrm{B}})\hat{\ }n_{turn}^{\mathrm{A}}$; Processes A and B are unchanged. The protocol rendering is still $(\mathrm{A} \,|\, \mathrm{B} \,|\, ReadyA^{false} \,|\, ReadyB^{false} \,|\, Turn^{A}) \backslash L$, where L is the set of all names and signals except **noncritA**, **critA**, **noncritB**, and **critB**, as before.

In the remainder of this section we prove Peterson's protocol correct, i.e. safe and live. We include the proof of safety for completeness, but concentrate on liveness.

Theorem 3 *Peterson's protocol is safe. In terms of Fig. 1, there is no reachable state where* A *and* B *have already executed lines ℓ_4 and m_4 but have not yet executed ℓ_6 or m_6.*

Proof. We follow the proof by contradiction of Peterson [31]. Suppose both processes succeed the test at ℓ_4 and m_4. Let A be the first to pass this test. At that time either *readyB* was false (meaning that Process B was between m_6 and m_2) or *turn* was set to A. In the first case, *readyA* will not be set to false before Process A leaves the critical section and *turn* is bound to be set to A by Process B before m_4 is executed. So the test at m_4 will fail. In the second case, since A is about to enter the critical section, *turn* cannot be set to B anymore and *readyA* is *true*, so again the test m_4 will fail for Process B. □

Peterson's protocol satisfies also the liveness property. As mentioned before, this result could not be proven for the formalisation of the protocol in CCS, assuming justness only.

Theorem 4 *Assuming justness, Peterson's protocol satisfies the liveness property: on each just path, each occurrence of* **noncritA** *is followed by* **critA** *(and similarly for* B*).*

Proof. Let π be a just path of the protocol. Since **noncritA** and **noncritB** are the only possible blocking actions, π must be $\{\mathbf{noncritA}, \mathbf{noncritB}\}$-just. If we get rid of all the restrictions we obtain a Y-just path

of (A | B | $ReadyA^{false}$ | $ReadyB^{false}$ | $Turn^{A}$) where $Y = \{\mathbf{noncritA}, \mathbf{noncritB}\} \cup L \cup \bar{L}_{\mathscr{H}}$. Suppose its decomposition π_A along Process A ends somewhere between instructions ℓ_1 and ℓ_4. Then the decomposition π_{ReadyA} along $ReadyA^{false}$ is also finite since only A can communicate with this process. Using the CCS rendering from Sect. 3 this statement would be incorrect, since there Process B can constantly interact with $ReadyA^{false}$, by reading its value; resulting in an infinite path $ReadyA^{false} \circlearrowleft n^{false}_{readyA}$.

By Def. 3, the path π_A must be X-just, and the path π_{ReadyA} Z-just, for sets $X, Z \subseteq Y$ with $X \cap \bar{Z}_{\mathscr{H}} = \emptyset$. Furthermore, $asgn^{true}_{readyA} \in Z$, since this action is enabled in the last state of π_{ReadyA}. Hence $\overline{asgn^{true}_{readyA}} \notin X$. Therefore π_A cannot end right before instruction ℓ_2. As a result, Process A is stuck either right before ℓ_3, or right before ℓ_4. In both cases Process B would not be able to pass the test before the critical section more than once. Indeed, in either case *readyA* is already set to *true*, thus Process B must use $turn = B$ to enter its critical section. But, if trying to enter a second time, it would be forced to set *turn* to A and will be stuck. When Processes A and B are both stuck, the path π is finite and an action τ or **noncritA** stemming from instruction ℓ_3 or ℓ_4 is enabled at the end, contradicting, through the first clause of Def. 3, the $\{\mathbf{noncritA}, \mathbf{noncritB}\}$-justness of π. □

8 Peterson's Algorithm for N Processes

In the previous section we presented an implementation in CCSS of Peterson's algorithm of mutual exclusion for two processes. In [31], Peterson also presents a generalisation of his mutual exclusion protocol to N processes. In this section we describe the algorithm and explain which assumptions should be made on the memory model in order for this protocol to be correct, for $N>2$. We claim that these assumptions are somewhat unrealistic.

A pseudocode rendering of Peterson's protocol is depicted in Fig. 2. In order to proceed to the critical section, each process must go through $N-1$ locks (*rooms*). The shared variable $room[i] = j$ indicates that process number i is currently in Room j. The shared variable $last[j] = i$ indicates that the last process to '*enter*' Room j is Process i. A process can go to the next room if and only if it is not the last one to have entered the room, or if all other processes are strictly behind it. This algorithm is also called the *filter lock* because it ensures that for all j, no more than $N+1-j$ processes are in rooms greater or equal than j. The critical section can be thought of as Room N.

A natural memory model, used in [25], stipulates that memory accesses from different components can overlap in time, and that a read action that overlaps with a write action of the same variable may yield *any* value. Extending this idea, we assume that when two concurrent write actions overlap, *any* possible value could end up in the memory. We argue that the algorithm fails to satisfy mutual exclusion when assuming such a model.

```
Process i  (i ∈ {1,...,N})
repeat forever
  ℓ1  noncritical section
  ℓ2  for j in 1...N−1
  ℓ3      room[i] := j
  ℓ4      last[j] := i
  ℓ5      await (last[j] ≠ i ∨ (∀k ≠ i, room[k] < j))
  ℓ6  critical section
  ℓ7  room[i] := 0
```

Figure 2: Peterson's algorithm for N processes (pseudocode)

Suppose there are three processes, A, B and C, and Processes A and B execute ℓ_1–ℓ_4 more or less simultaneously. When their instructions ℓ_4 overlap, the value C ends up in the variable *last*[*1*]—or any other value different from A and B. Hence they both perform ℓ_5, as well as ℓ_3–ℓ_4 for j=2. Again, the value C ends up in *last*[*2*]. Subsequently, they both enter their critical section, and disaster strikes.

It follows that Peterson's algorithm for N>2 only works when running on a memory where write actions cannot overlap in time, or—if they do—their effect is the same as when one occurred before the other. Such a memory can be implemented by having a small hardware lock around a write action to the same variable. This entails that one write action would have to wait until the other one is completed. A memory model of this kind is implicitly assumed in process algebras like CCS(S).

We show that, under such a memory model, Peterson's algorithm for N>2 does not satisfy liveness, unless we enrich it with an additional fairness assumption.

To prove this statement, let $N = 3$ and call the processes A, B and C. We show that (without the additional assumption) Process A can be stuck at ℓ_4 for j=1. Suppose Process A is at this line. Then A is about to set *last*[*1*] to *A*, but has not written yet. We can imagine the following scenario: Process B enters Room 1, and sets *last*[*1*] to *B*; then Process C enters Room 1, and sets *last*[*1*] to *C*. This allows B to proceed to Room 2, then to go in the critical section (because all other processes are still in room 1), and then to go back to Room 1, setting *last*[*1*] to *B*. This allows C to go to Room 2, to the critical section, and back to room 1, setting *last*[*1*] to *C*. Next B can enter the critical section again, etc. Hence Processes B and C can go alternately in the critical section without giving A a chance to set variable *last*[*1*]. (The variable is too busy being written by *B* and *C*.) This scenario cannot happen for $N = 2$ because after B sets *last*[*1*] to *B*, B is blocked until A sets it to *A*; so ℓ_4 will eventually happen (with progress as a basic assumption).

As a consequence, in order for Peterson's algorithm to be live for more than two processes, we must adopt the additional fairness assumption that *if a process permanently tries to write to a variable, it will eventually do so*, even if other processes are competing for writing to the same variable. This property appears to be at odds with having a hardware lock around the shared variable. Moreover, it cannot be implemented in CCSS assuming only justness: when two competitive processes try to write the same variable, nothing guarantees that both will eventually succeed.[6] As a result any CCSS-rendering of Peterson's algorithm for N processes does not possess the liveness property, unless one makes a fairness assumption. The problem comes from the fact that the variables *last*[·] are written by several parallel processes. Signals only allow a writer to set a variable while it is being read but do not allow multiple writers at the same time.

We believe that the problem does not come from a lack of expressiveness of CCSS but from the protocol, which, while not being incorrect in itself, requires a memory model that assumes write actions to happen eventually, even though simultaneous interfering write actions are excluded; whether this is a realistic assumption on modern hardware requires further investigation.

9 Lamport's Bakery Algorithm

In this section we analyse Lamport's bakery algorithm [25], another mutual exclusion protocol for N processes. It has the property that processes write to separate variables; only the read actions are shared. We give a model for this algorithm in CCSS and prove its liveness property, assuming justness only.

[6] Let us consider a CCSS process $(x^{true}|W_1|W_2)\backslash L$ where processes W_1 and W_2 are infinite writers ($W_i \stackrel{def}{=} \overline{asgn_x^{false}}.W_i$) and L is the set of communication names. A path where W_1 always succeeds, meaning that the decomposition along W_2 is empty, is just because the latter decomposition is $\{asgn_x^{false}\}$-just and all the other decompositions $\emptyset$-just.

Process i $(i \in \{1,\ldots,N\})$

repeat forever

ℓ_1 **noncritical section**
ℓ_2 $choosing[i] := true$
ℓ_3 $number[i] := 1 + \max(number[1],\ldots,number[N])$;
ℓ_4 $choosing[i] := false$
ℓ_5 **for** j **in** $1\ldots N$
ℓ_6 **await** $(choosing[j] = false)$
ℓ_7 **await** $(number[j] = 0 \vee (number[i], i) \leq (number[j], j))$
ℓ_8 **critical section**
ℓ_9 $number[i] := 0$

Figure 3: Lamport's bakery algorithm for N processes (pseudocode)

A pseudocode rendering of Lamport's bakery algorithm is depicted in Fig. 3. Lines 2–4 are called the *doorway* and lines 5–7 are called the *bakery*. In the doorway each process 'takes a ticket' that has a number strictly greater than all the numbers from the other processes (at the time the process reads them). The variable $choosing[i]$ is a lock that makes line 3, which is usually implemented by a simple loop, more or less 'atomic'. To ensure that the holder of the lowest number is next in the critical section, each process goes through a number of locks in the bakery (Lines 5–7). When process i enters the critical section, the value it has read for $number[j]$, if not 0, is greater or equal than its own $number[i]$, for all j.

We now model this algorithm in CCSS. As usual, we define one agent for every pair (variable, value). The variables *choosing* can take values *true* or *false*, and *number* any non-negative value. The modelling of a Boolean variable is addressed in Ex. 1, and for the integer variables we define:

$$number[i]^k \stackrel{def}{=} \Big(\sum_{l\in\mathbb{N}} asgn^l_{number[i]} \,.\, number[i]^l\Big) \hat{\ } n^k_{number[i]} \,.$$

Each process i begins with a non-critical section before entering the doorway.

$$P_i \stackrel{def}{=} \textbf{noncrit}[i] \,.\, \overline{asgn^{true}_{choosing[i]}} \,.\, doorway[i]^1_0$$

Line 3 encodes several read actions, an arithmetic operation, and an assignment in a single step. In CCS(S) (and most programming languages) this command is modelled by several atomic steps, e.g. by the simple loop $m := 0$; **for** j **in** $1\ldots N\{m := \texttt{max}(m, number[j])\}$; $number[i] := 1 + m$. We define processes $doorway[i]^j_m$ that represent the state of being in the doorway **for**-loop for a process i with loop index j and local variable m storing the current maximum.

$$doorway[i]^j_m \stackrel{def}{=} \Big(\sum_{k>m} n^k_{number[j]} \,.\, doorway[i]^{j+1}_k\Big) + \Big(\sum_{k\leq m} n^k_{number[j]} \,.\, doorway[i]^{j+1}_m\Big) \,,\; j \in \{1,\ldots,N\}$$

We then define $doorway[i]^{N+1}_m$, which represents the termination of the **for**-loop by

$$doorway[i]^{N+1}_m \stackrel{def}{=} \overline{asgn^{m+1}_{number[i]}} \,.\, \overline{asgn^{false}_{choosing[i]}} \,.\, bakery[i]^1_{m+1} \,.$$

The process $bakery[i]^j_m$ represents the state of being in the bakery **for**-loop for process i with loop index j and $number[i] = m$. For $j \in \{1,\ldots,N\}$:

$$bakery[i]^j_m \stackrel{def}{=} n^{false}_{choosing[j]} \,.\, \Big(n^0_{number[j]} + \sum_{k>m\vee(k=m\wedge j\geq i)} n^k_{number[j]}\Big) \,.\, bakery[i]^{j+1}_m \,.$$

Finally, $bakery[i]_m^{N+1}$ is the exit of the bakery **for**-loop, granting access to the critical section:

$$bakery[i]_m^{N+1} = \mathbf{crit}[i] \,.\, \overline{asgn^0_{number[i]}} \,.\, P_i$$

Our bakery algorithm is the parallel composition of all processes P_i, in combination with the shared variables $choosing[i]$ and $number[i]$, restricting the communication actions:

$$\Big(\Big| \underset{i \in \{1,\ldots,N\}}{} (P_i \mid choosing[i]^{false} \mid number[i]^0) \Big) \backslash L \,,$$

where L is the set of all names and signals except **noncrit**[i] and **crit**[i].

We now prove the liveness of (our rendering of) the algorithm, given that it is straightforward to adapt Lamport's proof of safety of the pseudocode [25] to CCSS. Since every process writes in its own variables, no process can be stuck because of concurrent writing. Therefore, the only possibility for a process (call it A) to be stuck is when trying to read a variable, so at ℓ_3, ℓ_6 or ℓ_7.

If Process A is stuck at ℓ_3, trying to read $number$[B] for some process B, B will get stuck at ℓ_6 for j=A, because $choosing$[A] remains *false*. So, Process B cannot be perpetually busy writing $number$[B], and A cannot be stuck at ℓ_3.

If A is stuck at ℓ_6, then from the point of view of A, some process B is all the time in the doorway. It follows from the argument above that B cannot be stuck in one visit to its doorway, so it must be a repeating series of visits. This is impossible because when A tries to read $choosing$[B] for the first time, the value of $number$[A] is set and will not change anymore, so if B goes back to the doorway, it is bound to set $number$[B] > $number$[A] and will not be able to enter the critical section anymore.

Suppose that Process A is stuck at ℓ_7. Any process B that enters the doorway will receive a $number$[B] strictly larger than $number$[A] and be stuck in the bakery. So if A is stuck, eventually all processes are stuck at ℓ_7, which is impossible since every finite lexicographically ordered set has a minimal element.

10 Conclusion, Related Work and Outlook

This paper presents a minimal extension of CCS in which Peterson's mutual exclusion protocol can be modelled correctly, using a justness assumption only. The signalling operator allows processes to emit signals that can be received by other processes. The signalling process is not blocked by the emission of the signal, which means that its actions are in no way postponed or affected by other processes reading the signal. This property is crucial to correctly model mutual exclusion.

Our process algebra, *CCS with signals*, is strongly inspired by, and can be regarded as a simplification of, Bergstra's *ACP with signals* [4]. The idea of a signal as a predicate on states, rather than a transition between states, stems from that paper. However, the non-blocking nature of signals was not explored by Bergstra, who writes "The relevance of signals is not so much that process algebra without signals lacks expressive power". This point is disputed in the current paper.

CCS with signals is not the first process algebra with explicitly non-blocking read actions. In [10] Corradini, Di Berardini & Vogler add a similar operator to PAFAS [12], a process algebra for modelling timed concurrent systems. The semantics of this extension is justified in [11]. They show [10] that this enables the liveness property of Dekker's mutual exclusion algorithm [13, 15], modelled in PAFAS, when assuming *fairness of actions*, and in [8] they establish the same for Peterson's algorithm, while showing that earlier mutual exclusion algorithms by Dijkstra [14] and Knuth [24] lack the liveness property under fairness of actions. Fairness of actions is similar to our notion of justness—although formalised in a

quite different way—except that all actions are treated as being non-blocking. The notion of time plays an important role in the formalisation of the results in [10, 8], even if it is not used quantitatively. Our process algebra can be regarded as a conceptual simplification of this approach, as it completely abstracts from the concept of time, and hence is closer to traditional process algebras like CCS and CSP.

The accuracy of our extension depends highly on which memory model is considered as realistic. It is well known that in weak or relaxed memory models, mutual exclusion protocols like Peterson's or the bakery algorithm do not behave correctly; when employing a weak memory model, mutual exclusion is handled on the hardware layer only—this is not covered here. An extremely plausible memory model allows parallel non-blocking writing, but admits any value being written when two parallel write actions overlap. This memory model is compatible with the bakery algorithm, and with Peterson's algorithm for two processes, but—as we show—not for Peterson's algorithm with $N \geq 3$ processes. Instead one needs a form of sequential consistency, assuming that parallel write actions, or a parallel read/write, behave as if they are executed in either order.

When postulating sequential consistency, it is plausible to assume some kind of mutual exclusion between write actions being implemented in hardware. This in turn allows the possibility of a write action being delayed in perpetuity because other processes are writing to the same variable. Similarly, read actions could be blocked by a consistent flow of write actions. A third type of blocking is that write actions can be obstructed by read actions. However, this kind of blocking is questionable; it could be that during a parallel read/write the write action wins, and only the read action gets postponed.

When assuming all three kinds of blocking, the CCS rendering of mutual exclusion protocols—illustrated in Sect. 3—is fully accurate, and by [20] we conclude that no such protocol can have the intended liveness property. When disallowing write actions being blocked by read actions, but allowing write/write blocking, we get the modelling in CCSS. Using CCSS, we verified the correctness of the bakery algorithm, and Peterson's algorithm for two processes, whereas Peterson's for $N > 2$ fails liveness. The latter protocol becomes correct if we assume sequential consistency without any kind of blocking. Whether this is a realistic memory model on modern hardware needs further investigation. Regardless, we conjecture that such a memory can be modelled in an extension of CCSS with broadcast communication, i.e. the combination of the process algebras presented here and in [21].

The liveness property of Dekker's algorithm, when assuming merely justness, or fairness of actions, requires not only non-blocking reading, but also that repeated assignments to a variable x of the same value cannot block the reading of x [10]. This assumption can be modelled in CCSS, by defining *readyA* of Ex. 1 by $\underline{x^{true}} \stackrel{def}{=} (asgn_x^{false} \,.\, x^{false})\hat{}\, n_x^{true}$ and $x^{false} \stackrel{def}{=} (asgn_x^{true} \,.\, x^{true})\hat{}\, n_x^{false}$, and replacing write actions $\overline{asgn_x^v}$ by $(\overline{asgn_x^v} + n_x^v)$. Alternatively, a pseudocode assignment $x := v$ could be interpreted as **if** $x \neq v$ **then** $x := v$ **fi**.

Although mutual exclusion protocols cannot be modelled in standard Petri nets—when not assuming fairness—[22, 33, 20], it is possible in nets extended with read arcs [33]. This opens the possibility of interpreting CCSS in terms of nets with read arcs, whereas an accurate semantics of CCSS in terms of standard nets is impossible. A read arc from a place to a transition requires the place to be marked for the transition be enabled, but the token is not consumed when the transition is fired. This behaviour really looks like signalling, so we conjecture that a read-arc net semantics of CCSS is fairly straightforward.

Finally, the definition of justness appears complicated because it includes the decomposition of paths. In order to compute if a path (an object from the semantics) is just or not just, we investigate the syntactic shape of the states on that path. It could be that the semantic object—the labelled transition system—is not well adapted to the problem of justness. Giving a semantics to CCSS that inherently includes the decomposition of paths—inspired by [6, 7, 1, 27]—could be an interesting idea for future research.

References

[1] L. Aceto (1994): *A Static View of Localities*. Formal Aspects of Computing 6(2), pp. 201–222, doi:10.1007/BF01221099.

[2] L. Aceto, A. Ingólfsdóttir, K. G. Larsen & J. Srba (2007): *Modelling Mutual Exclusion Algorithms*. In: Reactive Systems: Modelling, Specification and Verification, Cambridge University Press, pp. 142–158, doi:10.1017/CBO9780511814105.008.

[3] J. C. M. Baeten & C. Verhoef (1993): *A Congruence Theorem for Structured Operational Semantics with Predicates*. In E. Best, editor: Proc. *CONCUR '93*, LNCS 715, Springer, pp. 477–492, doi:10.1007/3-540-57208-2_33.

[4] J. A. Bergstra (1988): *ACP with Signals*. In J. Grabowski, P. Lescanne & W. Wechler, editors: Proc. Int. Workshop on *Algebraic and Logic Programming*, LNCS 343, Springer, pp. 11–20, doi:10.1007/3-540-50667-5_53.

[5] A. Bouali (1992): *Weak and Branching Bisimulation in Fctool*. Research Report RR-1575, Inria-Sophia Antipolis. Available at `https://hal.inria.fr/inria-00074985/document`.

[6] G. Boudol, I. Castellani, M. Hennessy & A. Kiehn (1993): *Observing Localities*. Theoretical Computer Science 114(1), pp. 31–61, doi:10.1016/0304-3975(93)90152-J.

[7] G. Boudol, I. Castellani, M. Hennessy & A. Kiehn (1994): *A Theory of Processes with Localities*. Formal Aspects of Computing 6(2), pp. 165–200, doi:10.1007/BF01221098.

[8] F. Buti, M. Callisto De Donato, F. Corradini, M. R. Di Berardini & W. Vogler (2011): *Automated Analysis of MUTEX Algorithms with FASE*. In G. D'Agostino & S. La Torre, editors: Proc. *GandALF '11*, EPTCS 54, Open Publishing Association, pp. 45–59, doi:10.4204/EPTCS.54.4.

[9] F. Corradini, M. R. Di Berardini & W. Vogler (2009): *Liveness of a Mutex Algorithm in a Fair Process Algebra*. Acta Informatica 46(3), pp. 209–235, doi:10.1007/s00236-009-0092-9.

[10] F. Corradini, M. R. Di Berardini & W. Vogler (2009): *Time and Fairness in a Process Algebra with Non-blocking Reading*. In M. Nielsen, A. Kucera, P. Bro Miltersen, C. Palamidessi, P. Tuma & F. D. Valencia, editors: Theory and Practice of Computer Science *(SOFSEM'09)*, LNCS 5404, Springer, pp. 193–204, doi:10.1007/978-3-540-95891-8_20.

[11] F. Corradini, M. R. Di Berardini & W. Vogler (2011): *Read Operators and their Expressiveness in Process Algebras*. In B. Luttik & F. Valencia, editors: Proc. *EXPRESS '11*, EPTCS 64, Open Publishing Association, pp. 31–43, doi:10.4204/EPTCS.64.3.

[12] F. Corradini, W. Vogler & L. Jenner (2002): *Comparing the worst-case efficiency of asynchronous systems with PAFAS*. Acta Informatica 38(11/12), pp. 735–792, doi:10.1007/s00236-002-0094-3.

[13] E. W. Dijkstra (1962 or 1963): *Over de Sequentialiteit van Procesbeschrijvingen*. Available at `http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF`. Circulated privately.

[14] E. W. Dijkstra (1965): *Solution of a problem in concurrent programming control*. Communications of the ACM 8(9), p. 569, doi:10.1145/365559.365617.

[15] E. W. Dijkstra (1968): *Cooperating Sequential Processes*. In F. Genuys, editor: Programming Languages: NATO Advanced Study Institute, Academic Press, pp. 43–112.

[16] J. Esparza & G. Bruns (1996): *Trapping Mutual Exclusion in the Box Calculus*. Theoretical Computer Science 153(1-2), pp. 95–128, doi:10.1016/0304-3975(95)00119-0.

[17] A. Fehnker, R. J. van Glabbeek, P Höfner, A. K. McIver, M. Portmann & W. L. Tan (2013): *A Process Algebra for Wireless Mesh Networks used for Modelling, Verifying and Analysing AODV*. Technical Report 5513, NICTA. Available at `http://arxiv.org/abs/1312.7645`.

[18] N. Francez (1986): *Fairness*. Springer, doi:10.1007/978-1-4612-4886-6.

[19] R. J. van Glabbeek (2011): *Bisimulation*. In D. Padua, editor: Encyclopedia of Parallel Computing, Springer, pp. 136–139, doi:10.1007/978-0-387-09766-4_149.

[20] R. J. van Glabbeek & P. Höfner (2015): *CCS: It's not Fair!—Fair Schedulers Cannot be Implemented in CCS-like Languages Even Under Progress and Certain Fairness Assumptions*. Acta Informatica 52(2–3), pp. 175–205, doi:10.1007/s00236-015-0221-6.

[21] R. J. van Glabbeek & P. Höfner (2015): *Progress, Fairness and Justness in Process Algebra*. CoRR abs/1501.03268. Available at `http://arxiv.org/abs/1501.03268`.

[22] E. Kindler & R. Walter (1997): *Mutex Needs Fairness*. Information Processing Letters 62(1), pp. 31–39, doi:10.1016/S0020-0190(97)00033-1.

[23] L. Kleinrock (1964): *Analysis of A Time-Shared Processor*. Naval Research Logistics Quarterly 11(1), pp. 59–73, doi:10.1002/nav.3800110105.

[24] D. E. Knuth (1966): *Additional comments on a problem in concurrent programming control*. Communications of the ACM 9(5), pp. 321–322, doi:10.1145/355592.365595.

[25] L. Lamport (1974): *A New Solution of Dijkstra's Concurrent Programming Problem*. Communications of the ACM 17(8), pp. 453–455, doi:10.1145/361082.361093.

[26] R. Milner (1989): *Communication and Concurrency*. Prentice Hall.

[27] M. Mukund & M. Nielsen (1992): *CCS, Locations and Asynchronous Transition Systems*. In R. K. Shyamasundar, editor: Proc. *FSTTCS '92*, LNCS 652, Springer, pp. 328–341, doi:10.1007/3-540-56287-7_116.

[28] J. Nagle (1985): *On Packet Switches with Infinite Storage*. RFC 970, Network Working Group. Available at `http://tools.ietf.org/rfc/rfc970.txt`.

[29] J. Nagle (1987): *On Packet Switches with Infinite Storage*. IEEE Trans. Communications 35(4), pp. 435–438, doi:10.1109/TCOM.1987.1096782.

[30] S. S. Owicki & L. Lamport (1982): *Proving Liveness Properties of Concurrent Programs*. ACM TOPLAS 4(3), pp. 455–495, doi:10.1145/357172.357178.

[31] G. L. Peterson (1981): *Myths About the Mutual Exclusion Problem*. Information Processing Letters 12(3), pp. 115–116, doi:10.1016/0020-0190(81)90106-X.

[32] A. Valmari & M. Setälä (1996): *Visual Verification of Safety and Liveness*. In M.-C. Gaudel & J. Woodcock, editors: Industrial Benefit and Advances in Formal Methods (FME'96), LNCS 1051, Springer, pp. 228–247, doi:10.1007/3-540-60973-3_90.

[33] W. Vogler (2002): *Efficiency of asynchronous systems, read arcs, and the MUTEX-problem*. Theoretical Computer Science 275(1-2), pp. 589–631, doi:10.1016/S0304-3975(01)00300-0.

[34] D. J. Walker (1989): *Automated analysis of mutual exclusion algorithms using CCS*. Formal Aspects of Computing 1(1), pp. 273–292, doi:10.1007/BF01887209.