Liveness analysis in process algebras
simpler techniques to model mutex algorithms

M.S. Bouwman
m.s.bouwman@student.tue.nl
Eindhoven University of Technology

January 31, 2018

Abstract

It has been established that standard process algebras are not able to correctly model the liveness properties of mutual exclusion protocols. It is necessary to either make a strong fairness assumption, which may not be realistic, or to adapt the (interpretation of) the language. In this paper we will analyze an earlier proposed extension to the semantics called signals and present a novel approach to solve the problem of correctly rendering liveness properties in process algebras that does not change the structural operational semantics.

1 Introduction

Rob van Glabbeek & Peter Höfner have shown that mutual exclusion protocols cannot be correctly implemented in CCS-like languages [1]. This goes against the widespread belief that any distributed system can be modeled in CCS-like languages. The authors have shown that mutual exclusion protocols cannot be modeled correctly without making strong fairness assumptions, which is unrealistic for the underlying memory model. In particular, liveness is not preserved when translating these protocols to process algebras. An assumption of justness should be enough to filter out paths that are unrealistic (and violate liveness properties), but this is not the case. So extensions are necessary to make it possible to correctly model mutual exclusion (mutex) protocols. Dyseryn, Höfner and van Glabbeek propose an extension with signals to solve the issues they discovered [2]. Their conclusions are based on their definition of justness.

In this work we will carefully analyze the argument for the necessity of a construct as for instance signals and see how signals solve the problem. We will particularly look at the Calculus of Communicating Systems (CCS), which is a process algebraic specification language. In CCS, Peterson's algorithm, a mutex protocol, will be analyzed as an example. The different types of fairness will be explained and it will be explained why paths that violate liveness are not excluded under the assumption of justness in CCS. After this has all been introduced, it will be examined how signals help to exclude the paths that lead to a liveness violation and the novel concept of signal actions will be introduced and analyzed. Signal actions are another way to address the same problems that signals address without the need of extra operators. These signal actions go against the conclusion of van Glabbeek and Höfner that CCS is not sufficient to model mutex algorithms correctly. A discussion on why these different conclusions are reached can be found in Section 11. We will also consider how signal actions could be integrated in existing toolsets for process algebras in Section 10.

2 Preliminaries - Calculus of Communicating Systems

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

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

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

Table 1: Structural operational semantics of CCS

The Calculus of Communicating Systems was introduced by Robin Milner [3] and can be used to model the behavior of concurrent systems using a simple but powerful syntax. It uses the sets $\mathscr{A}$ and $\mathscr{K}$ of *names* and *agent identifiers*, where names are used for actions and agent identifiers refer to processes. The set of *handshake actions* is defined to be $\mathscr{H} := \mathscr{A} \cup \bar{\mathscr{A}}$, where $\bar{\mathscr{A}} := \{\bar{a} \mid a \in \mathscr{A}\}$ is the set of *co-names* and where $\bar{\bar{a}} = a$. The complementary actions allow parallel processes to synchronize. Finally, $Act := \mathscr{H} \cup \{\tau\}$ is the set of *actions*, where τ is a special *internal action*. A central concept is that these actions may be performed resulting in a change of state, denoted by $P \xrightarrow{\alpha} P'$, where P and P' are states and α is some action. The syntax is defined by the following BNF grammar:

$$P ::= 0 \mid \alpha.P_1 \mid A \mid \sum_{i\in I} P_i \mid P_1|P_2 \mid P_1[f] \mid P_1\backslash L,$$

in order of occurrence:

- inactive process - 0 denotes the *inactive process* that is incapable of performing an action
- prefixes - the process $\alpha \,.\, P_1$ can perform an action α and continue as P_1;
- agent identifiers - $A \overset{def}{=} P_1$ specifies that the identifier A can be used to refer to process P_1, which may contain a reference to A again;
- choice - $\sum_{i\in I} P_i$ lets a process continue as some process P_j, where $j \in I$;
- parallel composition - $P_1|P_2$ denotes that P_1 and P_2 are executed in parallel;
- relabeling - $P_1[f]$ is the process P_1 where the actions are renamed by the function $f : \mathscr{H} \longrightarrow \mathscr{H}$, where $f(\bar{a}) = \overline{f(a)}$;
- restriction - $P_1\backslash L$, $L \subseteq \mathscr{H}$, is the process P_1 where execution of the actions in L is prohibited. The actions in L are only possible in the form of a synchronization of two parallel subprocesses, which only show up as τ's in the path over the entire process.

The usual notation for finite choices is using a +, the process $P_1 + P_2$ continues as either P_1 or P_2. The shorthand $(\alpha + \beta) \,.\, P$ can be used for $(\alpha \,.\, P) + (\beta \,.\, P)$. The structural operational semantics can be found in Table 1.

3 Preliminaries - Peterson's algorithm

To make things more concrete, we will analyze a specific well known mutual exclusion protocol: Peterson's algorithm. The correctness of this algorithm has been well established and the algorithm is easy to understand. The algorithm is given below in pseudocode. Intuitively, the two processes cannot both be in the critical section at the same time since the requirements to enter the critical section in the await statement cannot be satisfied at the same time. Intuitively, a process that wishes to enter the critical section will eventually be able to do so as it can independently move from its noncritical section to the await statement at which point the other process can do at most one more loop after which it is also stuck in the await statement and has set the turn to the other process.

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)

Here ℓ_6 is interpreted as process A entering the non-critical section and executing ℓ_1 is interpreted as process A leaving its non-critical section. Similarly for process B. This interpretation allows us to adopt the assumption that "atomic actions always terminate"[4] while allowing a process to stay in its non-critical section indefinitely.

A natural way to model Peterson's algorithm in CCS is to define process A and B as,

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

and to use the following paradigm to model the two-valued shared variables $turn$, $readyA$ and $readyB$ by CCS processes using x as the placeholder for the name of a variable:

$$x^{true} \stackrel{def}{=} asgn^{true}_{x} \,.\, x^{true} \;+\; asgn^{false}_{x} \,.\, x^{false} \;+\; \overline{n^{true}_{x}} \,.\, x^{true}\,,$$
$$x^{false} \stackrel{def}{=} asgn^{true}_{x} \,.\, x^{true} \;+\; asgn^{false}_{x} \,.\, x^{false} \;+\; \overline{n^{false}_{x}} \,.\, x^{false}\,.$$

In these processes $asgn^{val}_{x}$ represents a write action and n^{x}_{val} represents a read action. Peterson's algorithm is then described by the parallel composition:

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

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

The transition system contains actions τ, **noncritA**, **noncritB**, **critA** and **critB**. A distinction is made between blocking and non-blocking actions. Non-blocking actions are not dependent on input from the environment, whereas blocking actions are dependent on an external environment or synchronization with other processes. Actions τ, **critA** and **critB** are considered to be non-blocking. Actions **noncritA**, **noncritB** are considered to be blocking, a process may choose to never leave its non-critical section. These notions of blocking and non-blocking are critical in the analysis of liveness as will be shown in subsequent sections.

The two most important types of properties of any mutual exclusion protocol are safety and liveness. The safety property is that the processes competing for the critical section will never be in their critical section at the same time. The liveness property is that if a process wants to enter the critical section it will eventually be allowed to do so. In the case of Peterson's algorithm liveness is defined to be: each time process A performs **noncritA** it will execute **critA** within a finite number of steps, and, similarly for process B.

4 Progress, justness and fairness

There are different notions that describe levels of how fair a scheduler is. Progress means that a sequential process that can do some action will eventually do it. It ensures that the process $A \stackrel{def}{=} a.0$ will eventually do an a. Justness intuitively ensures the same for parallel processes that do not depend on each other. It ensures $A|B$, with $A \stackrel{def}{=} a.A$ and $B \stackrel{def}{=} b.0$, will eventually perform a b. Fairness on the other hand is a much stronger assumption, it ensures that $A \stackrel{def}{=} a.A + b.0$ will eventually perform a b. In general it is realistic to assume progress and justness but not fairness. Thus we want that all desirable properties of some system can be proven assuming only progress

and justness.

The properties progress, justness and fairness are defined on paths. A path is a sequence of alternating states and transitions: $P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} P_n \xrightarrow{\alpha_{n+1}} \ldots$. A path is deemed to be incomplete and therefore unrealistic if the path is not considered to be just. Paths can be decomposed to paths along subprocesses. By the operational semantics of CCS (see Table 1) there are three cases possible for the decomposition of a transition $P|Q \xrightarrow{\alpha} R$:

- 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 $R = P|Q'$.

A path π of a process $P|Q$ can then be decomposed into paths π_P and π_Q along P and Q, respectively, consisting of the concatenation of all actions stemming from P in π_P and all actions stemming from Q in π_Q. Similarly, any transition $P[f] \xrightarrow{\alpha} R$ can be decomposed as $P \xrightarrow{\beta} P'$, where $R = P'[f]$ and $\alpha = f(\beta)$. By decomposing each transition of a path of $P[f]$ the path of P is obtained. A decomposition of a path of $P\backslash L$ is defined likewise.

To be able to properly reason with the concept of justness we need a more formal definition that we can check on a given path. The following is the formal definition of justness as presented in [1].

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

1. a finite Y-just path ends in a state that admits actions from Y only;
2. 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 $\bar{Z} := \{\bar{c} \mid c \in Z\}$;
3. a Y-just path of $P\backslash L$ can be decomposed into a $Y \cup L \cup \bar{L}$-just path of P;
4. a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
5. 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.

This definition also captures progress as a finite path that can do a non-blocking action in its final state is not just.

5 Analysis Y-justness

5.1 When is a path Y-just?

The definition of Y-justness can be difficult to understand. We will therefore look at the intuition behind the clauses of the definition and explore a few simple examples.

The first clause states that a process that gets 'stuck' in some state after some finite path is Y-just with the actions that it can do in its final state in Y. If the process can do actions a and b in its final state, then it is $\{a, b\}$-just with the requirement that a and b are blocking. If a or b is non-blocking, the path is not just as it could have performed an action on its own. If a and b are blocking however, the path is just. An infinite path does not force us to put actions in Y. An infinite path along a process that does not contain parallel compositions is therefore $\emptyset$-just as there are no finite subprocesses.

Intuitively the second clause states that parallel processes that are both 'stuck' in some final state should not be able to communicate in their final states. Otherwise, this communication should be performed. Consider for example the processes $A|B$ with $A \stackrel{def}{=} c.A$ and $B \stackrel{def}{=} \bar{c}.B$ and the path of finitely many τ's. By clause 1, the paths along processes A and B are X-just, $c \in X$,

and Z-just, $\overline{c} \in Z$, respectively. This violates the second clause as $X \cap \bar{Z} \neq \emptyset$ and thus the path is not just. On the other hand, if one of the processes has other options than communicating then the communication may not happen. If $A \stackrel{def}{=} c.A + a.A$ and we consider the path of infinite a's and finitely many τ's then we will see that this path is just. The path along A is infinite and $\emptyset$-just and the path along B is $\{\overline{c}\}$-just. Now $X \cap \bar{Z} = \emptyset$ and since clause 2 also has the requirement that $Y \supseteq X \cup Z$ the path over the entire process $A|B$ is $\{\overline{c}\}$-just.

The third clause makes sure that actions in the restriction L do not show up in the Y-justness of the entire process. If we consider $(A|B)\backslash\{c\}$ with $A \stackrel{def}{=} c.A + a.A$ and $B \stackrel{def}{=} \overline{c}.B$ and the path of infinite a's and finitely many τ's, we will see that it is $\emptyset$-just. The decomposition is $Y \cup L \cup \bar{L}$-just which satisfies the requirement of the second clause that $Y \supseteq X \cup Z$.

The fourth clause simply ensures that renamings are dealt with. If we have a process $P[f]$, where f renames a to b, and a path that is $\{b\}$-just then that path is $\{a\}$-just along P.

The fifth and last clause reduces the justness of paths to the justness after on or more actions are taken. For example a path of the process $a.(A|B)$ is Y-just if it is Y-just along $A|B$ considering the suffix of the path without the leading a.

According to the definition a 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. Here it is important to realize that there are usually many sets Y that satisfy the requirements for Y-justness. In fact, the next subsection proves that it is monotonic. If, however, we are forced to put some action a in Y, then the path is a-enabled. Intuitively this means that from some point on the action a is enabled in every state but never performed without breaking justness. In two situations this can be the case: actions dependent on the environment may never happen (in Peterson's algorithm a process may choose to stay in its non-critical section) or there is some process that wants to perform a but it needs to synchronize with $\overline{a}$ while no other process ends in a final state in which it can do $\overline{a}$.

5.2 Monotonicity Y-justness

We prove that if some path π is Y-just then it is also Y' just if $Y \subseteq Y' \subseteq \mathscr{H}$. We show that this path is Y'-just by showing it satisfies all clauses of Definition 1 where we note that by definition of π being Y-just, the set Y satisfies all clauses.

Clause 1 only applies if π is finite. If it is infinite, then clause 1 cannot contradict our claim. If π is finite the clause requires that all actions possible in the final state are in Y for it to be Y-just. Since $Y \subseteq Y'$ we conclude that all these actions are also in Y'. If the path has been generated by a process of the shape $P|Q$, clause 2 applies. It requires that $Y \supseteq X \cup Z$ and $X \cap \bar{Z} = \emptyset$. Trivially $Y' \supseteq X \cup Z$. The requirement of $X \cap \bar{Z} = \emptyset$ does not influence Y or Y'. Furthermore X and Z do not need to be different and thus the decompositions are still X-just and Z-just respectively. If the path has been generated by a process of the shape $P\backslash L$, clause 3 applies. It cannot directly contradict our claim but requires that its decomposition is $Y \cup L \cup \bar{L}$-just. Since $Y \cup L \cup \bar{L} \subseteq Y' \cup L \cup \bar{L}$, our arguments can simply be applied on the decomposition. If the path has been generated by a process of the shape $P[f]$, clause 4 applies. Again there is no direct requirement that can contradict our claim but it does require that the decomposition is $f^{-1}(Y)$-just. Since $f^{-1}(Y) \subseteq f^{-1}(Y')$ our arguments can be applied on the decomposition. Clause 5 requires that each suffix of a Y-just path is Y-just. Since each suffix of the path is Y-just (by our assumption that π is Y-just) and clause 5 can only cause a problem if some other clause later down the path is violated, clause 5 cannot contradict our claim in itself. Hence, none of the clauses can contradict that π is Y'-just and we conclude that if a path is Y-just then it is also Y' just if $Y \subseteq Y' \subseteq \mathscr{H}$.

Note that we cannot just put anything in Y if we are looking at the element of a decomposition of a process. In particular if we are looking at processes that are in a parallel composition we need to be careful that the requirement $X \cap \bar{Z} = \emptyset$ of clause 2 is not violated.
Furthermore we note that it makes sense to try to prove a path Y-just for some Y that is as small as possible (remember that a 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) in order to exclude that it is enabled for some actions. It is not certain if there is always

a unique minimal set for Y.

6 Problems in CCS: Liveness violation

In this section we will analyze why the rendering of Peterson's algorithm is not satisfactory. In particular it will be demonstrated that there exists a path that violates the required liveness property but is considered to be just according to Definition 1.

Consider the path ρ where process A executes ℓ_1 and wants to perform $\overline{asgn_{true}^{readyA}}$. However, process B executes m_1 to m_4. At m_4 it forces $readyA^{false}$ to perform n_{false}^{readyA}, briefly disabling the action $\overline{asgn_{true}^{readyA}}$. After this, process A still needs to write but process B is quicker and performs m_5 to m_3 in a flash and sets the variable $readyA^{false}$ to false. This continues ad infinitum and process A never gets the chance to indicate that it wants to enter the critical section. Surely this path violates the liveness property.

We will show that ρ is an $\emptyset$-just path. If ρ is to be an $\emptyset$-just path of $(\mathrm{A} \mid \mathrm{B} \mid ReadyA^{false} \mid ReadyB^{false} \mid Turn^{A})\backslash L$ then it follows from clause 3 of Definition 1 that it suffices to show that ρ is a $L \cup \bar{L}$-just path along $(\mathrm{A} \mid \mathrm{B} \mid ReadyA^{false} \mid ReadyB^{false} \mid Turn^{A})$. According to clause 2 it then suffices to show that it is possible to decompose it into a S-just path along A, a T-just path along B, etc. with a requirement that $Y \supseteq S \cup T \cup U \cup V \cup W$ and a requirement that $X \cap \bar{Z} = \emptyset$, where $\bar{Z} := \{\bar{c} \mid c \in Z\}$ for all $X, Z \in \{S, T, U, V, W\}, X \neq Z$. The path along A is finite and is $\{\overline{asgn_{true}^{readyA}}\}$-just because of clause 1 of our definition of Y-justness. The paths along B, $ReadyA^{false}$, $ReadyB^{false}$ and $Turn^{A}$ are infinite and cannot be decomposed further so they are $\emptyset$-just. This satisfies the requirements on the composition so ρ is $\emptyset$-just and thus ρ is just.

Hence, justness is not enough to exclude paths that violate the liveness property in CCS.

7 Signals

7.1 Introducing signals

$$(P\hat{}s)\!\sim^{s} \qquad \frac{P \xrightarrow{\alpha} P'}{P\hat{}s \xrightarrow{\alpha} P'} \qquad \frac{P\sim^{s}}{(P\hat{}t)\sim^{s}} \qquad \frac{P_j\sim^{s}}{(\sum_{i\in I} P_i)\sim^{s}}\ (j \in I)$$

$$\frac{P\sim^{s}}{(P|Q)\sim^{s}} \qquad \frac{P\sim^{s},\ Q \xrightarrow{s} Q'}{P|Q \xrightarrow{\tau} P|Q'} \qquad \frac{P \xrightarrow{s} P',\ Q\sim^{s}}{P|Q \xrightarrow{\tau} P'|Q} \qquad \frac{Q\sim^{s}}{(P|Q)\sim^{s}}$$

$$\frac{P\sim^{s}}{(P\backslash L)\sim^{s}}\ (s \notin L) \qquad \frac{P\sim^{s}}{P[f]\sim^{f(s)}} \qquad \frac{P\sim^{s}}{A\sim^{s}}\ (A \stackrel{def}{=} P)$$

Table 2: Structural operational semantics for signals

As mentioned in the introduction, the use of signals is proposed to rule out paths violating the liveness property. The resulting language is dubbed CCS with signals or in short CCSS. A signaling operator $P\hat{}s$ is introduced to indicate that a process emits a signal. The predicate $P\sim^{s}$ is used to indicate that some process P emits a signal which may cause some process Q to receive the signal and change state as a result: $Q \xrightarrow{s} Q'$. The set $\mathscr{S}$ contains the names of all signals. The semantics of this new operator can be found in Table 2. For a more complete explanation of the reasoning behind signals and the properties of the operator we refer to the original paper [2]. In this paper we will simply look at the modified definition of Y-justness and how it rules out paths that violate the liveness property as they are considered unrealistic.

The following definition of Y-signalling paths gives an upper bound on the signals emitted in a path[2].

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

1. a finite Y-signalling path ends in a state that admits signals from Y only;
2. a Y-signalling path of a process $P \mid Q$ can be decomposed into an X-signalling path of P and a Z-signalling path of Q such that $Y \supseteq X \cup Z$;
3. 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;
4. a Y-signalling path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-signalling path of P;
5. and each suffix of a Y-signalling path is Y-signalling.

The notion of a just path is adapted to accommodate signals [2]:

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

1. a finite Y-just path ends in a state that admits actions from Y only;
2. 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 $\bar{Z}_{\mathscr{H}} := \{\bar{a} \mid a \in Z \cap \mathscr{H}\}$;
3. 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;
4. a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
5. 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.

7.2 Signals and Peterson's algorithm

To translate Peterson's algorithm to accommodate signals the following is done: process A and B remain exactly the same and the processes of the variables in our model are changed to fit the following template:

$$x^{true} \stackrel{def}{=} (asgn_x^{true} \,.\, x^{true} \;+\; asgn_x^{false} \,.\, x^{false})\hat{}\, n_x^{true}$$

Let us define ρ' to be a path similar to ρ where process A does not get the chance to write to *readyA*. We will see how we can prove $\overline{\rho'}$ is not just. Like before, the decomposition along A is finite and X-just, for some set X, and $\overline{asgn_{true}^{readyA}}$ must be in X. Since A is now the only process interacting with $readyA^{false}$, the path along $readyA^{false}$ is finite and by clause 1 of the definition of justness the path is Z-just, for some set Z, with $asgn_{true}^{readyA} \in Z$. However, the composition now violates clause 2 of the definition since $X \cap \bar{Z}_{\mathscr{H}} \neq \emptyset$. Intuitively, since the variable *readyA* is no longer interrupted by reads, the interaction between $readyA^{false}$ and A in the path violating the liveness property is continuously enabled and thus must eventually take place, assuming justness.

8 Signal actions

8.1 Introducing signal actions

Ideally, we would like to have a smaller change to the process-algebraic formalism than extending it with signals to be able to model mutex algorithms correctly. The challenge, it seems, is to exclude paths at which reading some shared variable prevents writing it. In systems were mutex algorithms are generally employed the central memory allows simultaneous reads and writes. It is desired that reads do not make a variable busy. This is exactly what the effect of signals is. If a variable is written finitely often but read infinitely often, then the path along the process of the

variable is finite as reads are modeled as signals that are only emitted finitely many times.
To achieve this result more directly a set of special *signal actions*, denoted by $\mathscr{R}$, is introduced. This set contains the sending read actions modeling processes of variables, $\{\overline{n_{false}^{readyA}}, \overline{n_{true}^{readyA}}, \overline{n_{false}^{readyB}}, \overline{n_{true}^{readyB}}, \overline{n_{A}^{turn}}, \overline{n_{B}^{turn}}\} \in \mathscr{R}$ for our model of Peterson's algorithm. We now change the first clause of Definition 1 which leads us to our new definition of justness:

Definition 4 The class of *Y-just* paths, for $Y \subseteq \mathscr{H}$, is the largest class of paths in $\mathscr{T}_{\text{CCS}}$ such that

1. a Y-just path that within finite steps reaches a state after which only actions from $\mathscr{R}$ or no actions at all take place, admits actions from Y only from all states after this state;
2. 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 $\bar{Z} := \{\bar{c} \mid c \in Z\}$;
3. a Y-just path of $P\backslash L$ can be decomposed into a $Y \cup L \cup \bar{L}$-just path of P;
4. a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
5. 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.

8.2 Restrictions on $\mathscr{R}$

Just changing the definition however is dangerous as we might break our concept of justness. Consider for example the following:

$$A \overset{def}{=} \overline{a} \,.\, A,\ B \overset{def}{=} b \,.\, B,$$
$$C \overset{def}{=} a \,.\, D \ + \ \overline{b} \,.\, C,$$
$$D \overset{def}{=} a \,.\, C,$$
$$(A|B|C)\backslash L,\ a \in \mathscr{R},$$

where L contains all actions.

The path over the entire process consisting of infinitely many τ's stemming from synchronizations on a and $\overline{a}$ and no τ's stemming from synchronizations on b and $\overline{b}$ is obviously just as no action is continuously enabled but never taken as the action b is only sometimes enabled. Our new definition of Y-justness however, does not consider it just as the path along C, which consists of infinitely many a's, reaches a state (the initial state) after which only signal actions are performed and is thus $\{\overline{b}\}$-just and the path along B is $\{b\}$-just which violates clause 2 of Y-justness as $X \cap \bar{Z} \neq \emptyset$. Hence we are restricting the possible traces too much, not considering all possible behavior.

We repair this by requiring that for every transition $P \xrightarrow{\alpha} P'$, $\alpha \in \mathscr{R}$, P is equal to P' otherwise α is not allowed to be in $\mathscr{R}$. In particular the actions in $\mathscr{R}$ may only occur in the shape $A \overset{def}{=} P \ + \ \alpha \,.\, A$. In our example we see that for $P \xrightarrow{a} P'$, P is not equal to P' as we lose the ability to perform $\overline{b}$. Intuitively, our signal actions only make sense if they do not alter the state.

When modeling a system using signal actions it important to be cautious in declaring an action a signal action because signal actions may not block other actions from happening (such as writing). This is an essential requirement when modeling a system as the model must represent the conditions in which the actual system operates. If reading would actually block another process from writing then it would be perfectly just if a variable is kept busy signalling its value. This requirement is not specific to signal actions (it is the same for signals) but a general warning to be cautious in the modeling phase to ensure that the assumptions on the memory model correspond to the environment in which the system will run.

8.3 Benefits signal actions

This approach has many similarities with the use of signals in that it targets the same actions in the original rendering of Peterson's algorithm in CCS and transforms them in some way such that they do not busy the process of the variable. Our extension only demands the specification of a set, containing actions that are already in the model, much like the sets of blocking and non-blocking actions. Furthermore the semantics of the language are not changed and existing models can be easily transformed.

8.4 Unjustness liveness violation

If we look back at the path that violates the liveness property we will see we get a similar situation as was achieved by using signals. The process is the same as in the original rendering in CCS with the addition of the definition of the signal actions $\mathscr{R}$, $\{\overline{n_{false}^{readyA}}, \overline{n_{true}^{readyA}}, \overline{n_{false}^{readyB}}, \overline{n_{true}^{readyB}}, \overline{n_{A}^{turn}}, \overline{n_{B}^{turn}}\} \in \mathscr{R}$. The path under consideration is ρ as introduced in Section 6, where process A executes **noncritA** but does not get the chance to execute $\overline{asgn_{true}^{readyA}}$ as process B loops through its code and writes to *readyA*.
The path along process A is finite and is thus X-just with the requirement that $\overline{asgn_{true}^{readyA}} \in X$ according to the first clause of Definition 4. The path along $readyA^{true}$ reaches a state after which only actions from $\mathscr{R}$ occur and thus the path is Z-just with $asgn_{true}^{readyA} \in Z$ by the first clause. The composition of the two processes violates clause 2 of Definition 4 since $X \cap \bar{Z} \neq \emptyset$ and thus ρ is not just.

The outcome of justness is the same for this path violating justness when using signals or signal actions. The paths are actually slightly different but very similar. The internal communications show up as τ's over the entire process for both methods. In the decompositions, however, there is a difference: using signals the signalling process does not contain an action (the path is empty) whereas when using signal actions, the signal action is in the path of the decomposition. The receiving end of a signal (action) has the same decomposition. The rest of the path and its decompositions are exactly the same.

We can go one step further and prove that in the rendering of Peterson's algorithm with signal actions, the desired liveness property is obtained: assuming justness, each occurrence of **noncritA** is eventually followed by **critA** (similarly for B).
Since **noncritA** and **noncritB** are the only possible blocking actions in the path over the entire process, any just path π is also {**noncritA**, **noncritB**}-just so it suffices to show that π is not {**noncritA**, **noncritB**}-just. By decomposing the restriction we derive that it is to be proven that π is $\{\textbf{noncritA}, \textbf{noncritB}\} \cup L \cup \overline{L}$-just along $A \mid B \mid ReadyA^{false} \mid ReadyB^{false} \mid Turn^{A}$. We examine how we can derive a contradiction if process A is able to execute **noncritA** but not **critA**. Due to the symmetry of A and B we only need to evaluate one of them. Suppose process A ends after **noncritA** and before $(n_{readyB}^{false} + n_{turn}^{A})$, the only case that would violate liveness of A. The path along *readyA* would then reach some state after which only signal actions happen since the path along A is finite and A is the only process that can cause *readyA* to perform an action that is not a signal action. By the second clause of Definition 4 the path along A must be X-just, for some set X, and the path along *readyA* must be Z-just, for some set Z, where $Y \supseteq X \cup Z$ and $X \cap \bar{Z} = \emptyset$. By the first clause $asgn_{readyA}^{true} \in Z$. Hence $\overline{asgn_{readyA}^{true}} \notin X$ and process A cannot end right before $asgn_{readyA}^{true}$. It must then be the case that it is stuck just before $asgn_{turn}^{B}$ or just before $(n_{readyB}^{false} + n_{turn}^{A})$. In both cases process B cannot pass the test to enter the critical section more than once since *readyA* is already set to true and it can only use n_{B}^{turn} to pass the test once, after which it sets *turn* to A. It will at some point get stuck, either because it no longer leaves the non-critical section or because it cannot pass the test to enter the critical section. When both process A and B are stuck then the path π would be finite and an action τ would be enabled stemming from $asgn_{readyA}^{true}$ or $asgn_{turn}^{B}$, which through the first clause of Definition 4 contradicts that π is a just path as τ is non-blocking.

9 Comparison signals and signal actions

Since the two approaches are so similar we might wonder how they compare and how they relate to each other. In essence, signals and signal actions are just different ways of modeling the same

phenomenon as there is a direct correspondence between them. In the following subsections it will be sketched how paths and processes can be translated from signals to signal actions and the other way around. It will also be argued that justness is preserved in these translations: the two ways of modeling coincide. These arguments are not formal enough to be considered a proof but do show how a proof might be constructed. Note that we cannot just prove that justness is preserved by proving that two processes using signals are bisimilar if and only if they are bisimilar after the translating the processes to signal actions, as justness is not preserved under bisimilarity.
We assume processes containing signals emit signals only at the base of the process, it has the shape $A \stackrel{def}{=} P \hat{} s$. If a signal is emitted in some process B at some other point than at the base then the part $P \hat{} s$ can be replaced with A. Since this can always be done and the resulting processes are equivalent we can assume all processes containing signals emit their signal at the base of the process without loss of generality.

9.1 Translating paths and processes

If we have a process containing signals then we can transform the process such that it no longer contains signals by introducing signal actions in the following way. It can be done by replacing each definition of a process of the shape $A \stackrel{def}{=} P \hat{} s$ with $A \stackrel{def}{=} P + (\overline{s} \ . \ A)$. The function $g(S)$ applies this transformation to the process S and all processes invoked by S. Translating a process containing signal actions to a process using signals can be done in an inverted way. Each definition of a process of the shape $A \stackrel{def}{=} P + (\overline{s} \ . \ A)$, $\overline{s} \in \mathscr{R}$, is replaced with $A \stackrel{def}{=} P \hat{} s$. The function $g'(S)$ applies this transformation to the process S and all processes invoked by S. Note that for any process S containing only signals and no signal actions, $g'(g(S)) = S$.

The translation of the path also needs to be considered. The function $f(\pi)$ translates a path π belonging to a process S containing signals to a path in $g(S)$. The function $f'(\pi)$ translates a path π belonging to a process S containing signal actions to a path in $g'(S)$. When translating between the two paradigms the path over the entire process stays the same as signals and signal actions only show up as τ's. We only need to consider the difference in decomposition. Table 3 shows the difference between handshakes and signals. The definitions are quite similar. For each occurrence of $P|Q \xrightarrow{\tau} P'|Q'$ in π caused by a signal or signal action, the decomposition needs to be considered. The receiving process has the same decomposition in both paradigms. When translating from signals to signal actions, the signalling process is empty for the decomposition of this action, so the signal action $P \xrightarrow{\overline{s}} P$ (if P is the signalling process) needs to be introduced, which is done by f. When translating from signal actions to signals the signal action $P \xrightarrow{\overline{s}} P$ needs to be eliminated from the path of the signalling process, which is done by f'.

$\dfrac{P \xrightarrow{a} P', \ Q \xrightarrow{\bar{a}} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'}$	$\dfrac{P \frown s, \ Q \xrightarrow{s} Q'}{P\|Q \xrightarrow{\tau} P\|Q'}$	$\dfrac{P \xrightarrow{s} P', \ Q \frown s}{P\|Q \xrightarrow{\tau} P'\|Q}$

Table 3: Comparison of structural operational semantics signals and handshakes

9.2 Justness preservation signals to signal actions

We will examine whether a just path π in a process S in the rendering using signals implies that the path $f(\pi)$ is just in $g(S)$.

We will not prove that justness is preserved in the general case but we will consider processes with a certain shape as this makes the proof much easier. We assume our process has the shape $S \stackrel{def}{=} (A|B|C...)\backslash L$, where L contains all handshake actions and signals and where $A, B, C...$ (and the processes that they may invoke) do not contain parallel compositions, renamings or restrictions. Note that Peterson's algorithm has this shape. Let π be some just path for S which means that it is Y-just for some set Y. By clause 3 of Definition 3, the decomposition along $A|B|C...$ is then V-just with $V = Y \cup L \cup \bar{L}_{\mathscr{H}}$. If $f(\pi)$ is to be just then it must be Y'-just for some set Y' and the decomposition of the restriction must be V'-just with $V' = Y' \cup L \cup \bar{L}$ by clause 3 of Definition 4.

It now suffices to prove that V-justness implies V'-justness.

If a path along a sequential process P in the parallel composition is finite in π then the path along P, π_P, is X-signalling and X'-just for some sets X and X'. The path $f(\pi_P)$ is then X''-just with $X'' = X \cup X'$. This is the case because if π_P is infinite it does not have an end state in $f(\pi_P)$ and is thus $\emptyset$-just/signalling in both paradigms (by the first clause of Definition 2, Definition 3 and Definition 4). In the case the path is finite in π then it has an end state in $f(\pi)$ as the added signal actions may still happen in the end state. Since they end up in the same state (as signals and signal actions do not change the state) and all regular actions that are enabled in one paradigm are also enabled in the other paradigm and the signals that are enabled imply there is a signal action enabled it holds that $X'' = X \cup X'$.

What is left is to bridge the gap between the sequential processes and the V/V'-just processes by considering the parallel composition. For every pair of processes P and Q in the parallel composition we know that π_P is X-just and X'-signalling, and π_Q is Z-just and Z'-signalling and $V \supseteq X \cup Z$, $X \cap \bar{Z}_{\mathscr{H}} = \emptyset$, $X \cap Z' = \emptyset$ and $X' \cap Z = \emptyset$ by the assumption that π is just and clause 2 of Definition 3. We know that $f(\pi_P)$ is X''-just and $f(\pi_Q)$ is Z''-just. Since $X'' = X \cup X'$ and $Z'' = Z \cup Z'$ we conclude that using signal actions the second requirement of clause 2 of Definition 4 is met: $X'' \cap Z'' = \emptyset$. Furthermore $V' \supseteq X'' \cup Z''$ as $V \supseteq X \cup Z$ and the extra signal actions that might be in X' or Z' are also in $\overline{L}$ which is a subset of V' so the first requirement is also satisfied.

This shows that a path that is just using signals is still just after translating the process and path to use signal actions.

9.3 Justness preservation signal actions to signals

This time we assume that we have a just path π in process S using signal actions and we will examine whether $f'(\pi)$ is just in $g'(S)$. We make the same assumption on the shape of the process as before.

By assumption π is Y-just for some set Y and it is therefore V-just, $V = Y \cup L \cup \overline{L}$ along $A|B|C...$ (after decomposing the restriction) by clause 3 of Definition 4. It is then to prove that the path containing signals is V'-just for some set V', $V' = Y' \cup L \cup \overline{L}_{\mathscr{H}}$ along the decomposition of the restriction, which would imply that the process is Y'-just over the entire process by clause 3 of Definition 3.

If the path π_P along a sequential process P in the parallel composition has an end state then π_P is X-just for some set X and ends in state after which only signal actions take place (clause 1 of Definition 4). The path $f'(\pi_P)$ is then finite as in the end state only some signal may be admitted. The path $f'(\pi_P)$ is then X'-just where $X' = X \backslash \mathscr{R}$ as the signal action is no longer possible and it is X''-signalling, where $X'' = X \cap \mathscr{R}$ as any signal action in X is converted to a signal (clause 1 of Definition 3). Thus $X = X' \cup X''$. If π_P does not have an end state then $f'(\pi_P)$ is infinite and $X = X' = X'' = \emptyset$.

Again what is left is to bridge the gap between the sequential processes and the V/V'-just processes by considering the parallel composition. For every pair of processes P and Q in the parallel composition, we know that the path π_P is X-just and the path π_Q is Z-just. By the assumption of π being just and clause 2 of Definition 4: $V \supseteq X \cup Z$ and $X \cap \bar{Z} = \emptyset$. Since $X = X' \cup X''$ and $Z = Z' \cup Z''$ we can conclude $X' \cap \bar{Z}'_{\mathscr{H}} = \emptyset$, $X' \cap Z'' = \emptyset$ and $X'' \cap Z' = \emptyset$. Furthermore, $V' \supseteq X' \cup Z'$ as $V \supseteq X \cup Z$ and $V \backslash V' = \overline{L} \backslash \overline{L}_{\mathscr{H}} \subseteq \mathscr{R}$, which are not in X' or Z'. The requirements of clause 2 of Definition 3 are then satisfied.

This shows that a path that is just using signal actions is still just after translating the process and path to use signals.

9.4 Conclusion of comparison

The previous sections reason that, assuming some shape for our process, any path in one paradigm is just if and only if that path is just after translating to the other paradigm.

10 Challenges in implementing liveness analysis

As has been shown mutex protocols modeled in typical process algebras do not have some desirable liveness properties, which can be fixed with some extension like signal actions or signals. The question remains how to analyze liveness properties and how to filter out paths that are not just or that do not make progress. Furthermore, there might be some restrictions on what kind of conversion/preprocessing steps a tool can use.

10.1 Manual analysis

Although it is not ideal, it is for smaller processes very feasible to do liveness analysis by hand. As seen by the example of Peterson's algorithm it is rather straightforward to identify all the paths that would violate liveness and it is not hard to prove whether these paths are just or not. For larger processes this might be more tedious and ideally we would not like to burden a system modeler with the concept of Y-justness if it would also be possible with a simple tool that does the analysis.

10.2 Specifying liveness properties

There are countless languages that can be used to specify properties of the behavior of a system. These properties can then be checked for a model in for example CCS or other process algebras. Most of these languages that specify properties of the behavior are able to describe liveness properties by asserting that eventually something good will happen. In the case of Peterson's algorithm we would have the requirement that after executing **noncritA**, **critA** will be executed within a finite number of steps (and a similar requirement for B). These languages already exist and are able to express what we desire so there is no issue with the current specification languages. The issue is that current tools evaluate all possible paths, including paths that are not just (paths without progress are generally not evaluated).

10.3 Checking requirement on $\mathscr{R}$

Since we have a syntactic requirement for the signal actions, a tool that incorporates signal actions should check this requirement. It could easily do a syntactical analysis of a model to verify that a signal action is only performed in a choice at the base of a process and immediately recurses to the same process, in other words that it has the shape $A \stackrel{def}{=} P + (\alpha \,.\, A)$, $\alpha \in \mathscr{R}$.

10.4 Preprocessing in tools

Tools usually convert the process specification to some intermediate format before checking a behavioral property. This intermediate format may be a linear process specification, a labeled transition system or some other format. In this intermediate format it is usually the case that any parallelism is removed, there is no distinction between different processes. This is problematic if the goal is to filter out unjust paths but not unfair paths. This is the case because, as seen in Section 4, behaviorally equivalent processes may have different just paths. Process $P \stackrel{def}{=} a \,.\, P + b \,.\, P$ need not ever perform a b assuming justness but not fairness where $Q \stackrel{def}{=} A|B$, with $A \stackrel{def}{=} a \,.\, A$ and $B \stackrel{def}{=} b \,.\, B$, is guaranteed to perform a b. This means that removing the notion of independent processes gives rise to problems in the analysis.

10.5 Filtering unjust paths

The big question is how unjust paths can be filtered out. Given some path it should be straightforward to determine whether that path is just. An approach to do this would be to construct a tree of the process, where the root is the entire process which in turn has children representing the decomposition of it. This can be done recursively. In this tree we could then determine the existence of end states in all the nodes and check the conditions for all the parent-child pairs. An approach to filter unjust paths would then be to compute all paths violating a specified liveness property and determine for each one whether it is just or not. It is unclear whether all paths

violating liveness can be computed in an efficient manner.

Another possibility might be to determine whether a liveness property holds by considering the shape of the labeled transition system (or other low level representation), where the transitions are annotated with knowledge about the process in which they occur and whether it concerns a signal action. Any path that is not just either ends in a state that admits a non-blocking action or is a loop in the LTS, in which in every state some non-blocking action (for example a synchronization between processes) is possible and where the process(es) that can perform this non-blocking action only perform signal actions in the loop. The liveness property is then violated if there is a finite path to some state in which no non-blocking action is possible or if there is a path to some just loop. If the process cannot get stuck in such a state or loop then, assuming the LTS has finitely many states, it must eventually come in the desired state. Other strategies might also be possible in LTS's and other low level representations as long as some annotation is added to preserve the information that an action stems from some process. This is all very speculative and should be further investigated.

It is a possibility that the labeled transition system and many other traditional representations of the behavior of systems are simply not well suited for liveness analysis.

11 Conclusions and future work

This paper shows an alternative for the analysis of liveness in process algebras, using a justness assumption only. The benefits of the signal actions are clear, they provide an easy way for a modeler to analyze liveness and have a smaller impact on existing process algebras, in particular the structural operational semantics remain unchanged.

It is curious that signal actions allow correct renderings of mutex algorithms without fairness assumptions considering the conclusions of van Glabbeek and Höfner in [1]. It is to be stressed that their conclusion is valid: assuming their definition of justness, it is impossible. What is remarkable is that they did not discuss how they arrived at their definition of justness or discuss alternatives. It is possible that they considered other definitions but had reasons to not use them. It is especially remarkable that they did not discuss constructs in which read actions are treated in some special way in the definition of justness (as is the case with signal actions) considering that they do make a distinction in blocking and non-blocking actions and incorporate this in the definition. Their work is still valuable as they show the importance of (non)-blocking actions, introduce a formal definition of justness for process algebras and prove that read actions can cause problems in liveness analysis. This paper simply shows that to address these problems it is also possible to incorporate read/signal actions in the definition of justness itself.

There is still much work to do before liveness analysis in process algebras can be widely adopted. At this point in time, analysis can already be done by hand but automated analysis via toolsets would be ideal. Research needs to be done on algorithms that are able to do the analysis (in an efficient way). There is also legwork needed to prove certain properties of our signal actions. In particular the reasoning in Section 9 needs to be converted to a formal proof with inductive definitions of the functions that convert the processes and paths. It also remains to be investigated whether our requirements on the actions in $\mathscr{R}$ can be relaxed so that a signal action is also allowed if the state is changed to a bisimilar state. Lastly, the definitions of justness and signal actions need to be translated to other process algebras than CCS that may have richer concepts like data and multi-actions.

We conclude that many steps are still necessary to make liveness analysis in process algebras a mature field. This makes it an exciting research field as many discoveries might be made. We hope that this paper has made a contribution to the further advancement of this field.

References

[1] Rob J van Glabbeek and Peter Höfner. CCS: It's not fair! *Acta Informatica*, 52(2-3):175–205, 2015.

[2] Victor Dyseryn, Rob van Glabbeek, and Peter Höfner. Analysing mutual exclusion using process algebra with signals. *arXiv preprint arXiv:1709.00826*, 2017.

[3] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[4] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. TOPLAS. 4:455–495, 07 1982.